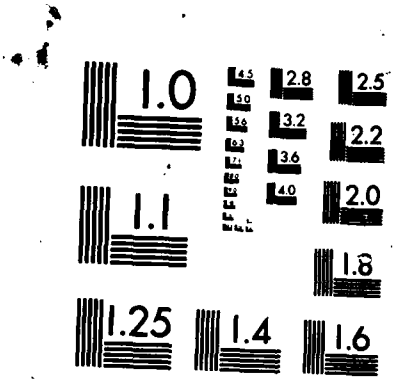
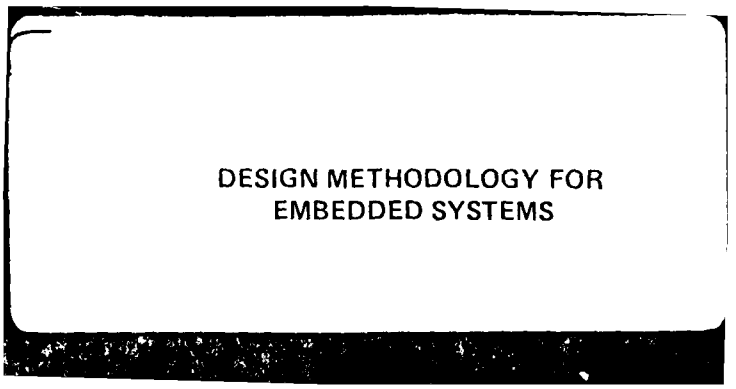
AD-A165 497 DESIGN METHODOLOGY FOR EMBEDDED SYSTEMS(U) ANALYTICS 1/2
INC WILLOW GROVE PA D LEFKOVITZ 15 NOV 85 TR-1500.21
N62269-80-D-0025

UNCLASSIFIED F/G 9/2 NL




DESIGN METHODOLOGY FOR
EMBEDDED SYSTEMS

(2)

Technical Report 1500.21

DESIGN METHODOLOGY FOR EMBEDDED SYSTEMS

Final Report

15 November 1985

Submitted to:
Naval Air Development Center
Warminster, PA 18974

Contract No. N62269-80-D-0025

David Lefkovitz

Analytics Inc.
Willow Grove, PA 19090

TABLE OF CONTENTS

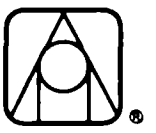
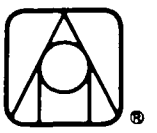
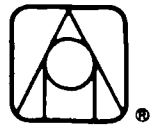


TABLE OF CONTENTS (continued)



LIST OF FIGURES

LIST OF FIGURES (continued)



TABLES

Distribution Statement A is correct for this report.
Per Dr. Roger Lee, NADC/Code 5033

1. INTRODUCTION

This contract has explored the multidimensional problem of the development of embedded software systems. In its reports and presentations the work has characterized these problems in various ways, but to view them as multidimensional is helpful inasmuch as the problems may be separated into different solution domains. These have been identified as: [1,2]

1. Communication
2. Principles and practices
3. Technology
4. Compiler (Ada) language structure and capability.

1.1 SOFTWARE DEVELOPMENT AS A PROBLEM IN COMMUNICATION

Figure 1-1 illustrates the communication problem. From the specification of requirements through maintenance of the operational system, people of different technical disciplines and possibly skill levels must interact and communicate. Creating appropriate linguistic and documentation forms is the major problem here.

The chain of communications, as shown in the figure, is as follows:

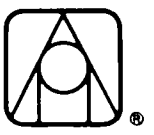
From: Technical (usually engineering), non-software expert

To: Software expert -- analyst

To: Software expert -- designer

To: Software expert -- programmer

Ideally, the loop is ultimately closed with final system testing, wherein all disciplines are mutually engaged. Then the system goes into



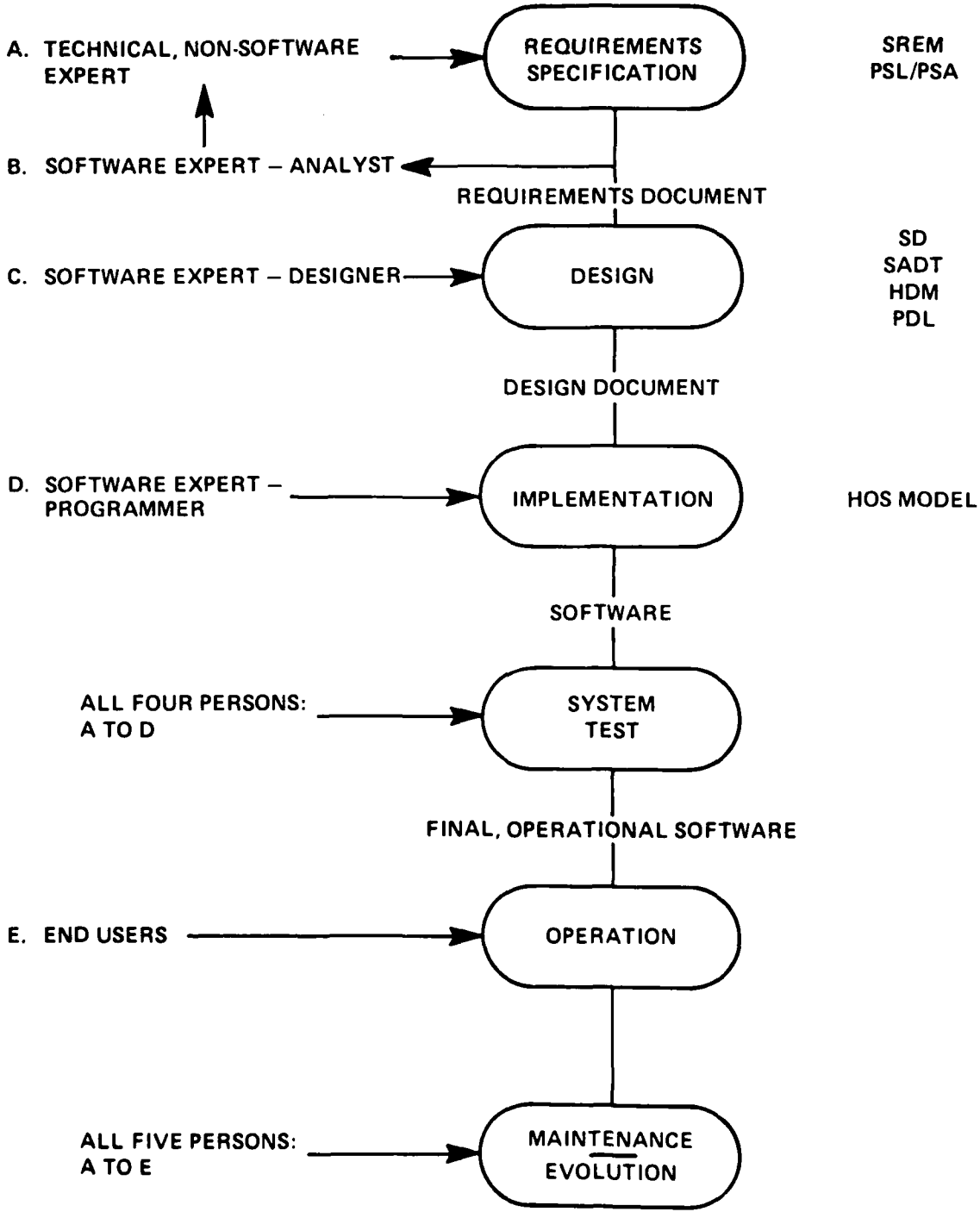


Figure 1-1. Personnel Types and Communication Requirements through the Life Cycle Stages

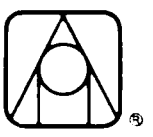
operation, but its maintenance (called evolution by some) may again involve all four types of people.

The requirements specification starts with statements by a technical, non-software expert. These are normally persons who are expert in one or more engineering disciplines associated with the systems within which the software (and computer hardware) is embedded. Their thought processes are therefore oriented toward their own engineering discipline, not software. They will interact with the software analyst or designer, who interprets their discipline-oriented requirements. The analyst's main purpose is to clarify these statements and make them less ambiguous.

The right side of the figure shows certain methods and automation aids at the various stages. The utility of these products must be evaluated in two ways. One is to determine whether they actually save labor (or produce significant, qualitatively superior results) in their immediate stage of application; second is to determine whether they provide downline labor saving (e.g., in maintenance) through the facilitation of those subsequent processes or in reduction of errors committed at those stages.

SREM [3] and PSL/PSA [4] are tools for the first development stage. They perform a number of important functions. First, they enable the system to be organized conceptually into distinct processes. Second, they enable any desired functionally descriptive format (including natural language) to be applied to each process. Third, they enable both control and data relationships among the processes to be clearly expressed. Fourth, they perform certain consistency checks based upon these relationships to indicate whether the specification is complete and error-free within the scope of the methodology.

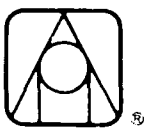
The second stage in the figure is design. Ideally, the requirements document contains all of the information needed by another expert, the software designer, to conceive and document the implementation design. This is a conceptual process that converts the requirements, in the form of mathematical



formulas, time constraints, information displays or reports, and input/output signal relationships, to design statements, in the form of data structures, computational algorithms, process modularization, process control (including multitasking), and logical decision formulas (usually in the form of tables or charts).

The Structural Analysis and Design Technique (SADT) [5] is a methodology that can be applied in the design of embedded systems. There are a number of such techniques that are used in commercial data base and transaction systems, such as the Jackson [6], Warnier-Orr [7], Structured Design [8], and SADT methods, but the SADT and Structured Design are better suited for embedded systems. These methods enable the modular and intermodular control and data flow to be expressed, and, to a limited degree, some of the control logic, but they are not applicable to the detailed definition of data structures and to the expression of algorithms, nor are they properly developed for intertask synchronization. The designer will improvise in these areas by using traditional flowcharting, decision table methods or even natural language. A more structured textual approach that combines the rigor of a flowchart with the flexibility of natural language has recently evolved to fill this need. It is called a program design language (PDL) [9].

At the next stage, implementation, a third software expert, the programmer, enters. This person is responsible for coding and debugging. There is also an overlap area, called detailed program design, between the designer and programmer, which often causes problems when not well-defined. It may involve the detailed definition of data structures such as arrays, list structures, and data types. It may also involve the lowest level of iterations, establishment of iteration limits, details of I/O operations, and the handling of errors and exceptions. Whether the designer or programmer performs these lowest-level design tasks depends upon such factors as the ability, cost, and interest of the respective person. A designer without prior programming experience would normally not do as good a job as a programmer with a good sense of design. Designers are usually paid more than a programmer and therefore,

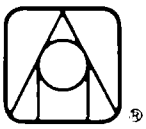


given equal ability, it would be less costly for the programmer to do it. And finally, some designers eschew such detailed work ("I might as well write the program"), while others not only enjoy it but feel that their total concept of the design may not be properly implemented unless they provide this lowest level of detail as well.

Regardless of who does it, the PDL is emerging as an effective means for documenting this design detail.

The next stage involves system tests that should bring all participants together to close the entire loop. Testing should be incremental, from small groups of networks or subsystems up through the entire system. The system then becomes operational, after which follows a stage that is called maintenance or evolution. The former term refers to correction of problems reported by the users; the latter to modifications in the existing design to accommodate additional needs.

The relative cost in each of these stages can vary widely according to the skill of the personnel involved and, to a lesser extent, the nature of the problem; however, it has been found that better quality (which usually implies more time and cost) in an earlier stage reduces the cost of all subsequent stages. Thus, a clear statement of requirements will require less definitional time for the designer and can avoid the very costly problem at system test of actual requirement. Similarly, a clear and sufficiently detailed design presents an unambiguous translation of the requirements to the programmer. Also, it usually results in a more efficient implementation than if the design is at a higher level and the programmer fills in with a more detailed design. Obviously, excessive cost at an earlier stage may not reduce the total development cost; hence we have an optimization problem in which the proper balance of time, cost, and quality are desired at each stage. The key factors, as presented in the discussion of Figure 1-1, are: (1) the ability of persons in the four different disciplines to communicate effectively and for the results of one

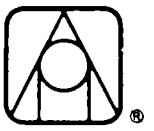


stage to be communicated accurately to the next, and (2) the availability of powerful tools at each stage to improve both the quality and speed of each of these persons.

The methods thus have two principal purposes. One is to improve the inherent performance of the technical activity within a stage. The other is to facilitate communication between the four disciplines as the development progresses through the life cycle stages.

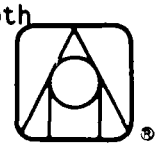
The solution to the *communication* problem is generally thought to be found in *methodology*. That is, if linguistic and documentational structure can be imposed, both within and between these various disciplinary groups, then two benefits will follow. One is that concepts can be better focused, at the appropriate disciplinary level, and the other is that error, through misunderstanding, can be reduced. These are valid principles, but this investigator believes that current approaches toward methodology development are lacking in two ways. First, insufficient attention is paid to the details of what these various disciplines do in the technical performance of their tasks. Instead, methodologies tend to proceed from a set of general principles and end up with something that either is inconsistent with normal and natural practice at the detailed operational level or remains at too high a level of definition, leaving implementation, at the detailed level, to individual practitioners. The latter is not as bad as the former, and, in fact, may be helpful in the long run, because it provides the opportunity to test many different specific implementations, based on the same set of principles. It would remain, then, to systematically evaluate them, at an appropriate time, because ultimately a standard will be required.

The second criticism is that a clearer idea of the differences among the terms *methodology, method, and tool* is needed, because these actually affect the way in which a methodology is developed.



The dictionary defines methodology as "The system of principles, practices, and procedures applied to any specific branch of knowledge." Method is defined as "a means or manner of procedure; especially a regular and systematic way of accomplishing anything." In summary, methodology is the system (i.e., some coordinated and integrated set) of procedures, while method is the detailed definition of each individual procedure in the methodology. Left unresolved is the scope of the procedure, analogous to a block-structured computer program. That is, the entire program is a procedure; hence the entire methodology could consist of one method. Alternatively, a program may have any number of internal or external procedures. Similarly, a methodology may consist of any number of methods within the methodological system. The key point here is that the detailed elucidation of the procedure is the method; the general description of functions and the systematic integration of methods is the methodology. The programming analogy to a methodology is the functional description of each procedure and the call tree that depicts their calling relationship; the analogy to a method is the flowchart or structured logic description of the procedure.

Of specific interest here, the methodology would define the procedures required to fulfill the two major functions of: (1) improvement in the inherent performance of the technical activity within a stage; and (2) facilitation of interdisciplinary and interstage communication. The method would define the mechanisms whereby each of the procedures is to implement that function. These mechanisms can assume two forms. One is a set of procedural instructions that present a sequence of operations and some formal approach to the stage activity. It may also include some other formal aids, such as tabular or diagrammatic constructs, which are, in effect, precursors to the second mechanism -- tools. A tool is a highly formal or even mechanized device that manipulates or transforms information. That is, it renders it from one linguistic form to another, where the first (which we can call either the input or the operand of the tool) is generally less structured and possibly ambiguous, and the second (which we can call the output) is more highly structured and either less ambiguous or totally unambiguous. Examples of tools are text editors, tables, flowcharts, block diagrams, and a DBMS. More sophisticated tools that involve elaborate and complex transformations of information are simulators and compilers. Both



mechanisms of methods -- procedural instructions and tools -- must play a role in achieving the dual goal of the methodology.

We have thus defined the following bottom-up hierarchy of concepts:

(1) Tools

(2) Procedural Instructions

(3) Method

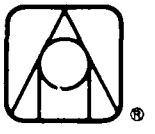
(4) Methodology

The question now asked is: Can we best achieve the dual goal by a top-down approach that starts with methodology and ends with tools or by a bottom-up approach that starts with tools and ends with methodology?

This brings us to the second problem dimension.

1.2 SOFTWARE DEVELOPMENT AS A PROBLEM IN PRINCIPLES AND PRACTICES

Although the conventional approach is to develop methodology first and then tools, it is suggested here that it may be more advantageous to reverse this process. Figure 1-2 shows that the designer as well as the programmer of software applications is subjected to a number of influences. Five of these are shown in an order (from A to E) that progresses from greater to lesser conceptual generality. General principles (A) include such concepts as topdown design, structured design, modifiability, and reusability. Principles of software engineering (B) include modular programming, abstraction, typing, cohesion, and information hiding. Traditional and ad hoc design/programming practice (C) represents personal or organizational styles and approaches that have accumulated through experience and particular avenues of education. Implementation language factors (D) are essential for program design, but there are two schools of thought regarding the higher-level system design and design methodology in particular. This investigator believes that implementation language should be taken into account at every stage of design.



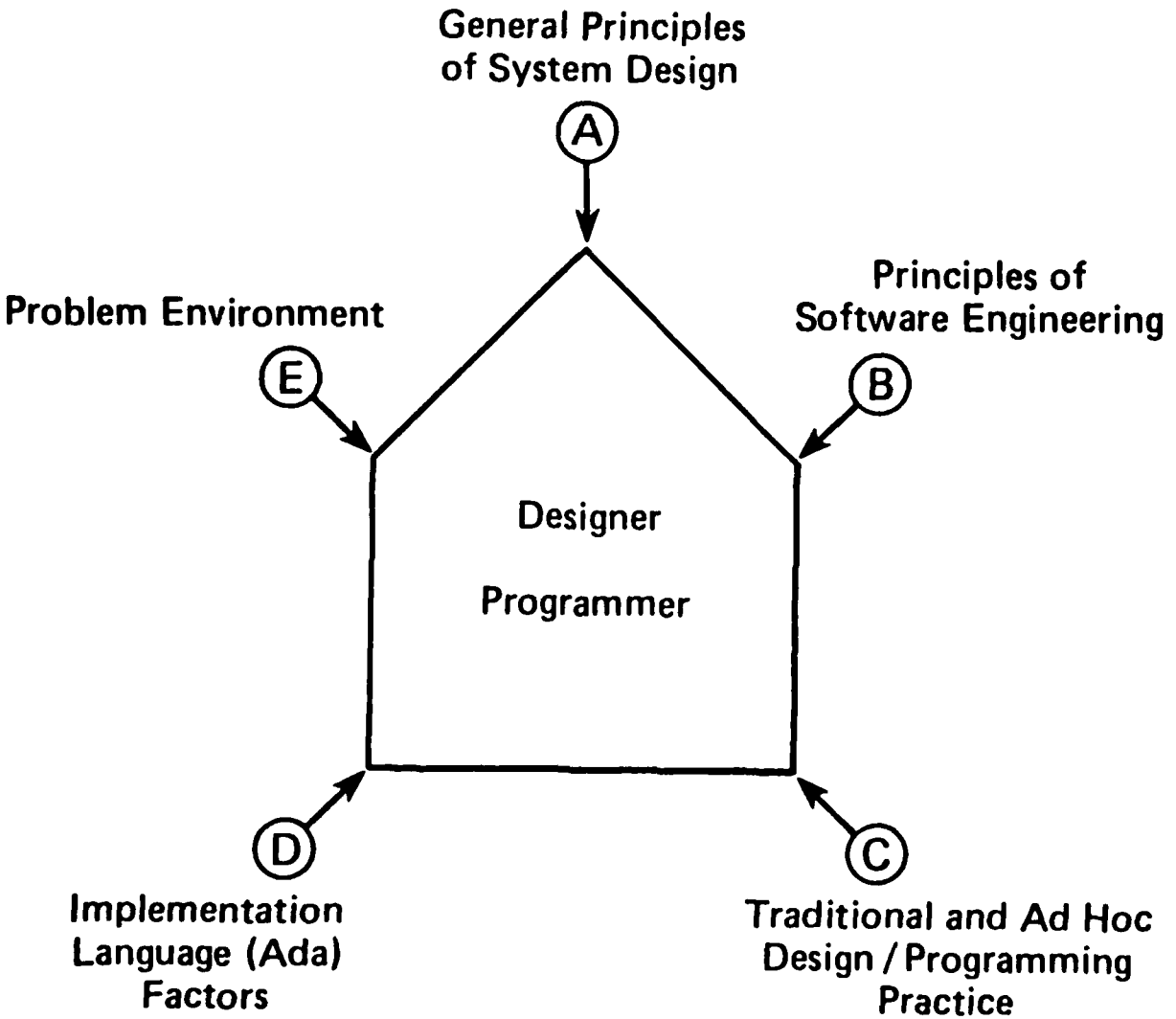


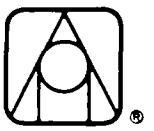
Figure 1-2. The Methodology Development Environment

The situation is somewhat analogous to structural engineering, wherein the designer of a bridge must presume knowledge of construction methods, tools, and materials. Finally, the problem environment (E) obviously affects the design specifics, but may or may not influence the method of design or the tools. For the purpose of this discussion, three broad classes of problem environment can be identified. The first is the data base and transaction-oriented environment; the second is the real-time, embedded system environment; and the third is the non real-time, computation-intensive environment. The three have somewhat different demands on hardware, software, and the design approach. The data base/transaction systems require large-scale secondary memory with intensive I/O activity to these stores. They usually require large main memory stores for data buffering, but can take advantage of virtual memory to permit large programs or time-sharing. They do not normally require multi-tasking.

Embedded systems typically do not permit large-scale secondary storage (i.e., electromechanical rotating devices) and accommodate program size to solid-state main memory because of the operating environment constraint. They almost invariably involve some kind of parallel processing, which is implemented either via multitasking or a simple round-robin, time-sliced executive.

The non real-time, computation-intensive systems require very fast processors, and although they can utilize secondary storage and virtual memory, they are not always practical because of the ultra high-speed computational requirement.

What is emerging here is that, although these problems have their own domains or dimensions in some conceptual problem space, there are significant functional relations between them that must be thoroughly examined and understood (for example, the issue of whether or not and how to integrate a particular technological component that may change both in time and application into a methodology). Thus, should the Ada language be so integrated, and if so,



where and how? At the specification stage? The preliminary design stage? The detailed design stage?

The reverse question is also important: How should methodology be developed that is flexible enough to admit currently changing and newly emerging technologies, such as multi-distributed processing and artificial intelligence?

1.3 SOFTWARE DEVELOPMENT AS A PROBLEM IN TECHNOLOGY

The design of software takes place within an environment that transcends the specific functions or algorithms in the specification. Just as the designer of a bridge must consider its total environment, so too must the software designer. In the case of the bridge, there is the access road system, residences in the area, and possibly water traffic. In the case of software, there are three major environmental factors:

(1) Application Environment

(2) Higher-Level and Control Software Environment

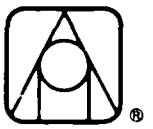
(3) Hardware Environment

1.3.1 Application Environment

One way to characterize embedded system software applications is to divide all software into two broad classes: real-time and non real-time. Real-time software has a critical requirement to interact with and respond to a physical environment within which it is embedded; non real-time software does not. Embedded systems fall into the first class. The emphasis here is on the word *critical*. A simple test is to say that the total system fails if the software fails to meet necessary time response requirements. Of course, "system failure" may have a subjective interpretation. For example, how does one classify an interactive data base-and transaction-oriented system, such as for travel reservations or banking?

The response of the data base system is usually load dependent and can vary widely from a pre-specified desired response. A design response time may be established as two seconds, but the point of failure is subjective because the consequence of exceeding this response time is to increase customer dissatisfaction but not necessarily to deny eventual service. Only if the response time were to degrade to the point of causing customers to leave the service queue could one begin to say that the system had failed, and even here the failure is only partial because some customers would leave and others would forebear and stay. Another important factor that would distinguish an interactive data base/transaction system from an embedded real-time system is that the opportunity to improve response time usually exists, in the former case, by the addition of more hardware -- memory, processors, channels, and disk drives. This is usually not the case in the latter. In this same regard, it is usually the case that program storage and program execution space are the same and are both finite and limited in embedded systems. Large-scale storage, like disks, often cannot be configured; consequently, the entire program is stored and executed from solid-state, non-rotating storage. Hence, commonly used time-sharing techniques like swapping and virtual memory are inapplicable. Finally, most, though not all, non real-time applications perform a single algorithm or procedure per transaction, or (at worst) a sequential series of algorithms, and it is left to the time-sharing operating system to switch the CPU from processing one transaction (i.e., user) to another, based upon the relatively simple criterion of FIFO queue management. Time criticalness in CPU servicing of the queue is usually ameliorated by the fact that a majority of the total (wall clock) algorithm process time is spent in waiting for disk access. If the CPU queue should become too large, then, as stated above, more hardware will help, up to a point.

By contrast, real-time software almost invariably requires that different algorithms run concurrently and interact by passing input and output data to each other.



In summary, the principal distinguishing characteristics of embedded software that are especially important to the designer of such systems are:

(1) The entire program usually must be stored in and be executed from the same internal, solid-state memory.

(2) From an external viewpoint, processes run in parallel and must communicate in a control and/or data transfer sense.

(3) Internally, the parallel processes may either be executed in true concurrency on multiple processors, or they may be executed in some mode that shares a single processor. There are two approaches to such time-sharing. One is called the cyclic task executive [10], wherein each process (or task) is assigned a fixed time slot on a round-robin queue that is cyclically (and synchronously) executed. The second method is called multitasking, wherein an operating system function maintains a list of all tasks and their service status, and transfers control to a given task on the list, based upon various criteria that are built into the operating system.

1.3.2 Software Environment

By this is meant all support software between the application program and the hardware. There may be several layers of such software. At the top is the operating system. Below this may be certain major utility functions such as sorts and file management. These do not generally apply to embedded systems. Another major piece of software under the operating system, which does apply, is the compiler or assembler. Thus, three types of software may appear in this environment: operating system, utilities, and compiler. This report will also discuss another kind of utility that can operate under the compiler, but still be above the application, and hence may be considered to be a part of this environment. Its purpose is to enhance or extend certain compiler functions that are useful from a designer's viewpoint, but are not directly available in the compiler.

1.3.3 Hardware Environment

The following hardware configuration factors will affect the design:

(1) Number of processors

(2) Type of memory -- RAM, RAM plus disk, ROM

(3) Size of memory

(4) Speed of processors

(5) Types of I/O channels and interrupt systems

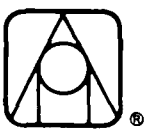
(6) Processor configurations -- central vs. distributed

This study has focused upon certain aspects of these three environments. In the application, it has focused upon parallel processing and its implications for establishing tasks, task priorities, and inter-task communication. In the software, it has focused upon the use of the Ada compiler and has explored some of the possibilities of utility enhancement by means of a software mechanism called the Virtual Real-Time Machine. In the hardware, it has assumed that multitasking is still required because of a need to time-share the processor.

1.4 SOFTWARE DEVELOPMENT AS A PROBLEM IN COMPUTER LANGUAGE AND CAPABILITY

The relationship of the computer language to the specification and design problem was mentioned in Section 1.1 above. However, the language is of central importance to the implementation and maintenance stages in the development lifecycle. The language must balance four factors. One is to provide the programmer with the broadest possible range of hardware capabilities. The second is to provide a high level language that will relieve the programmer of as much coding detail as possible. The third and fourth factors are not a function of the language design itself but of administration. Third is that the language should be portable to multiple host systems and translatable into the object code of multiple target systems. Fourth is that as few languages as possible (ideally one) be used in system development. The adoption of Ada by DoD for embedded systems development is directed toward this last point. This work has, therefore, focused upon Ada as a development language.

Ada has some structural complexities and subtleties that most of its predecessors do not. For example, modularity in languages like FORTRAN, ALGOL, and COBOL is implemented via the procedure. In PL/1 another type of module, though still viewed syntactically as procedure, is the task. Ada programs can



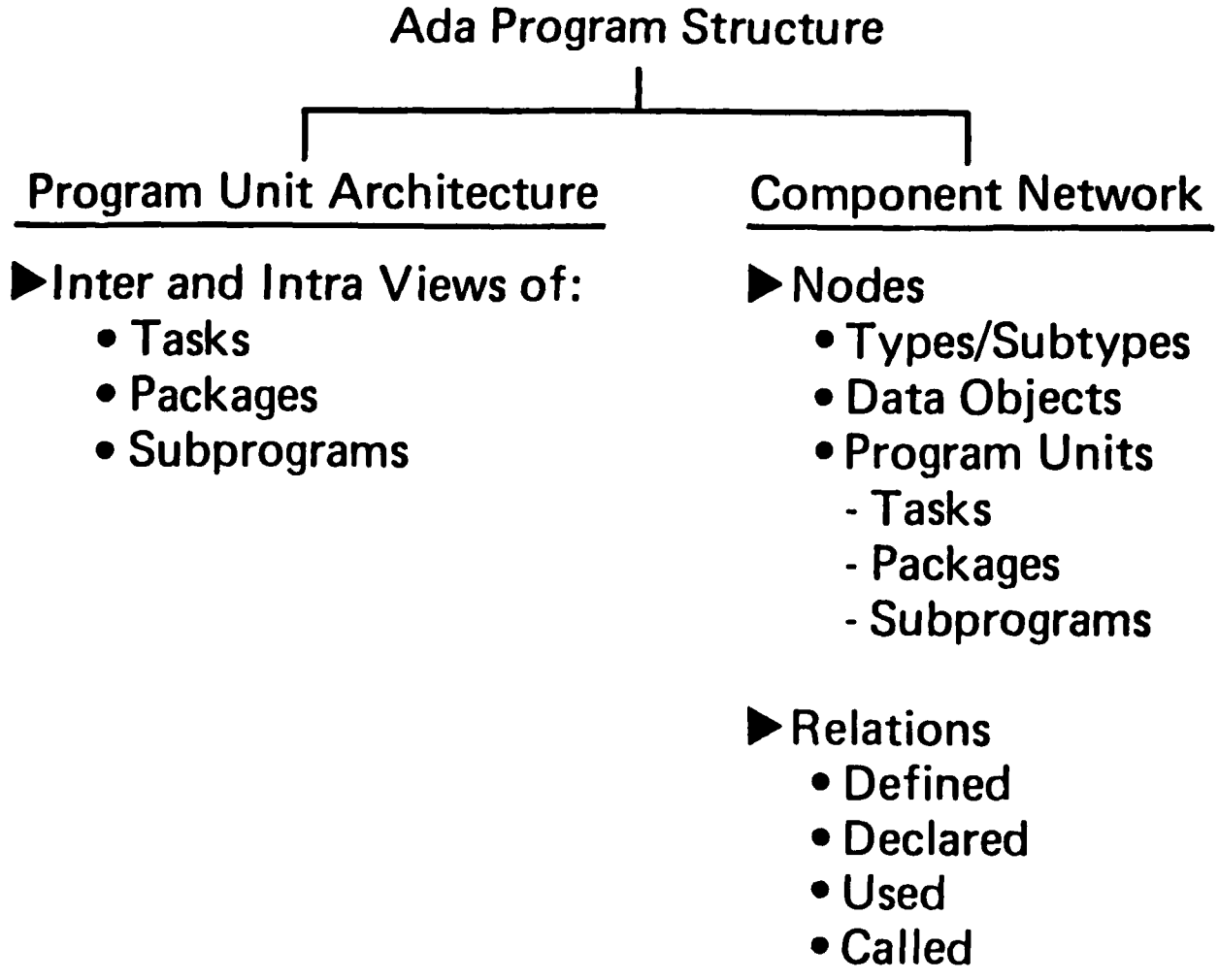


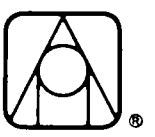
Figure 1-3. Two Views of Ada Program Structure

85-21-04-FR

have a more elaborate structure. Figure 1-3 presents Ada program structure in two ways. First is the structure or architecture of the three classes of Ada program units: tasks, packages, and subprograms. These units have a complex structure both within individual class and in combination. As individual classes, one can conceive of six architectural "views" of an Ada program. These are the inter- and intratask, inter- and intrapackage, and inter- and intra-subprogram views. Although these views along pure unit class lines are of value, they do not reveal the full complexity of the program design and power of the language, which is shown by the right side of Figure 1-3, the component network. Here the language is considered to have five distinct node types in a conceptual network. A few others could also be defined but are of lesser importance, and the result may be more confusing than helpful. The five major node types are: types/subtypes, data objects, and the three program units (tasks, packages, and subprograms). These are all interrelated by means of four relations, which in network terminology would be represented as branches. These are "where defined, declared, used, and called."

Another example of the component network is the relationship among program units. Figure 1-4 shows how each program unit relates to others of the same or of a different class, as well as to the two other network components (types and data objects). The Intra view reveals not only the complex internal structure of these components but also the cross-component relationships.

A task has both a visible and a hidden part. The visible parts are the entries and the select structure that controls them. The hidden is a select structure and a substructure of procedural statements which, in turn, consist of calls to other tasks, calls to subprograms within the task, and the procedural code itself. The task calls may either be simple calls or may be embedded in a select structure. The purpose of the select structure is to enable alternative actions to a non-response to the call, such as a fixed waiting period (including zero) or a logical condition on acceptance of the call.



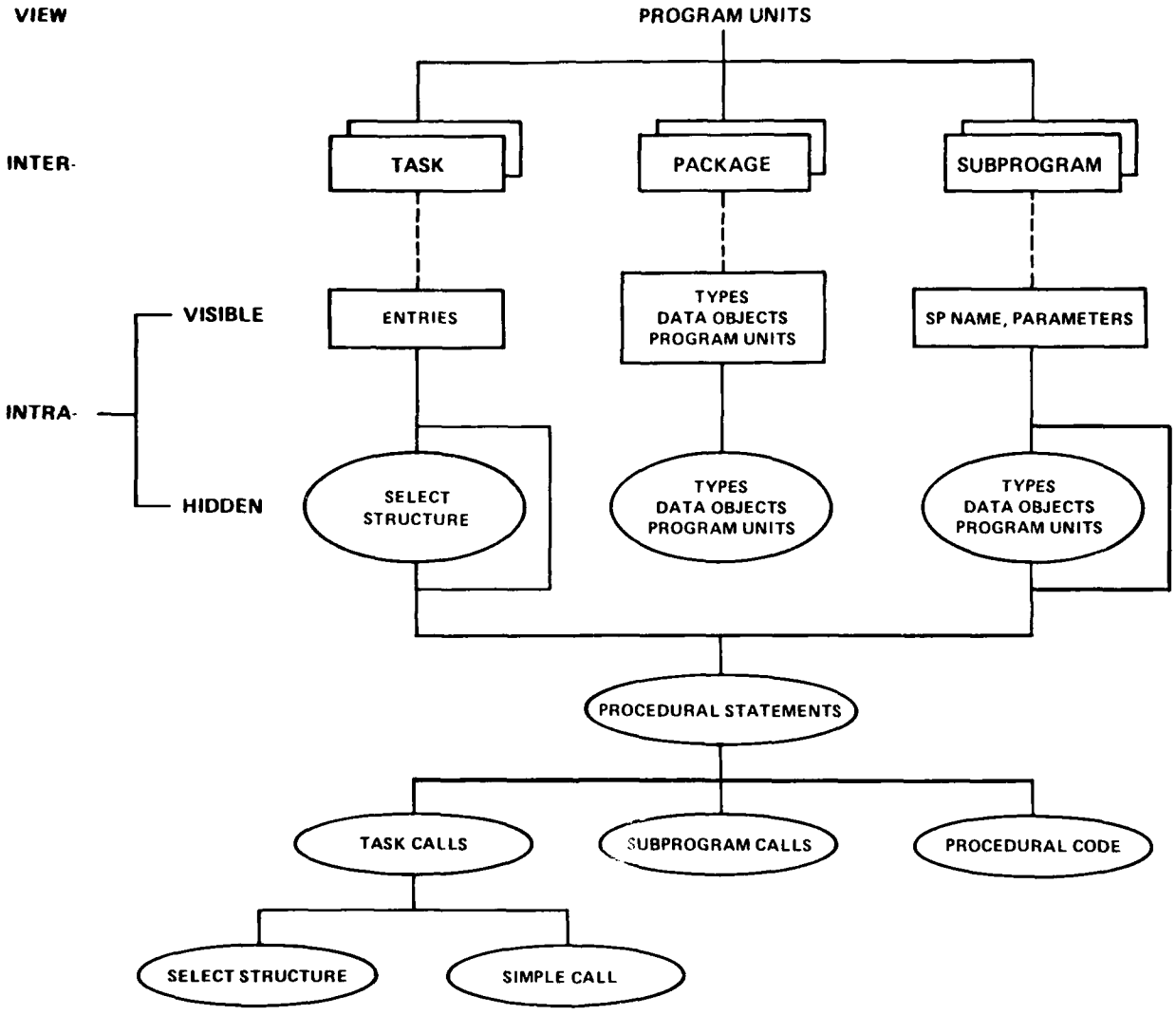


Figure 1-4. Detailed View of Component Network

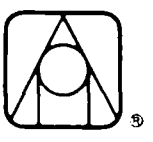
The visible or the hidden part of a package may be any of the five network components. There are no procedural statements associated with the package per se, only with the tasks and subprograms appearing within it. Hence, a package is not a formal part of the program logic in that it is never "called." As the name implies, it is only a conveyance for the five network components.

The subprogram has the conventional visible part of a name and parameter list. Hidden are the procedural statements that are formally identical to those of a task.

Thus, one may ask the following kinds of questions:

1. What subprograms are declared within a given task, package, or subprogram?
2. What is the call tree structure within a given subprogram to other subprograms?
3. Expand question 2 by including task calls.
4. In which program unit is a given package used?
5. Which subprograms of package X are used in program unit Y?
6. What program units are declared in the hidden part of a given package?
7. What packages are used by a given task?
8. What data objects are declared in a given program unit?
9. What is the scope of a given data object?

Figure 1-5 presents a detailed view of the Ada task and intertask communication structure. It also presents, for contrast, certain non Ada provided capabilities that are conceptually possible and would be design-limiting by the Ada language per se. These will be discussed further in a later section. Figure 1-5 shows that the task has four main characteristics: it has a priority (in relation to other tasks), it is activated, it communicates with other tasks, and it is terminated.



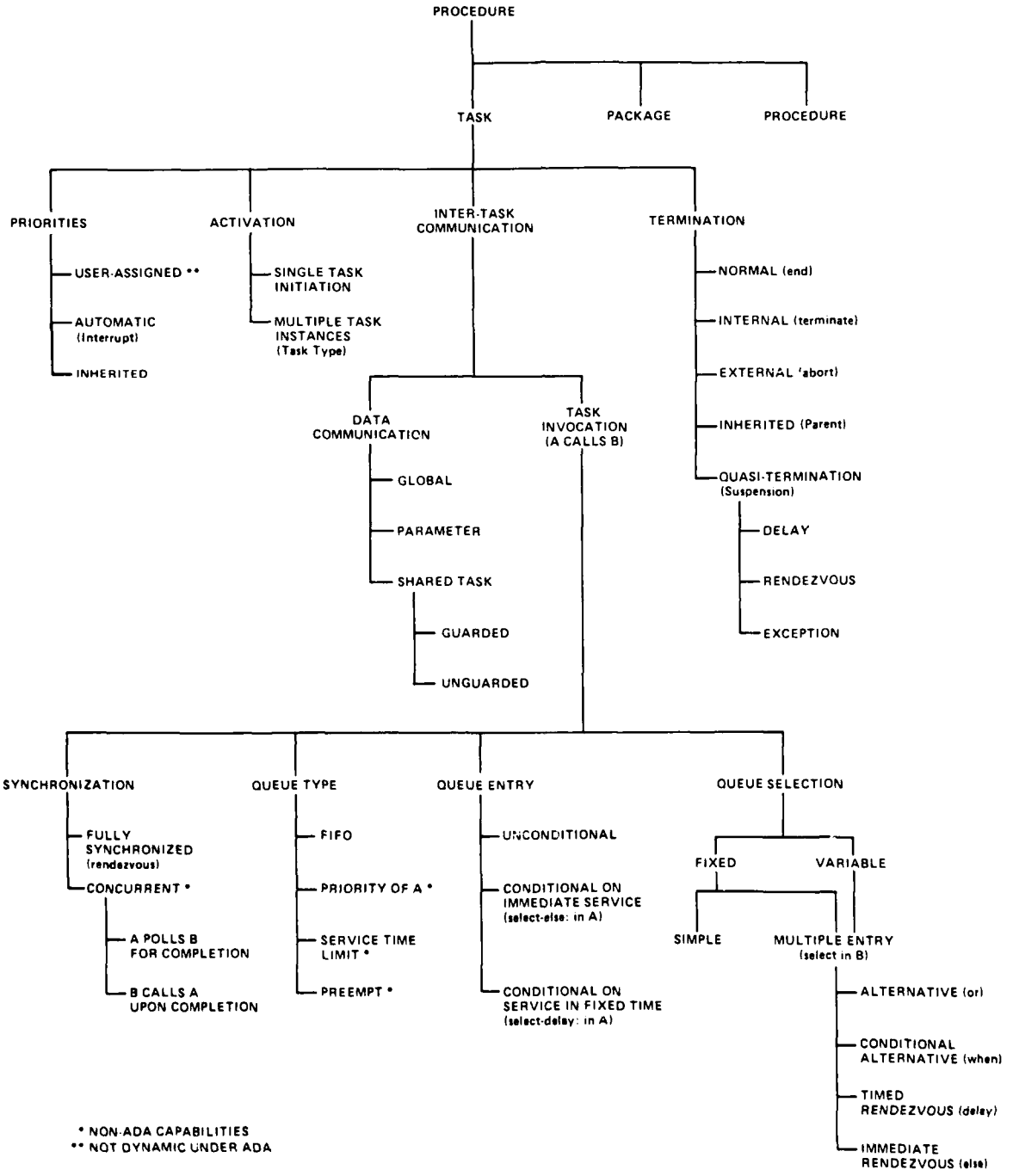


Figure 1-5. CPB Analysis of Parallel Processing Under ADA

1.4.1 Priority

The task priority is an artificiality. In a real parallel processing machine, tasks actually run and execute in parallel. In a virtual or simulated parallel processor, tasks execute in a sequence determined by an executive program, based upon several factors, one of which is a priority assigned to the task, relative to others. Tasks that are in a "running" state are selected for execution based upon the relative values of their priorities.

In Ada, task priorities are assigned by the programmer at compile time. Ada does not permit them to be re-assigned under program control at run time, unlike PL/1. This will preclude the writing of programs that can adapt responsiveness to conditions that arise while the program is running. However, a called task will inherit the priority of the calling task while it is servicing the call of that task. An interrupt in Ada is also viewed as a task entry, but it is automatically assigned the highest priority.

1.4.2 Activation

Though tasks run in parallel, they do not necessarily have to be in an active or run state throughout the entire operation of the program. In addition to the time and method of activation, there is the concept of a multiple task instance or a task clone. This is the analog in multitasking to re-entrancy. A given task can be activated multiply, and in all instances will run in parallel. In Ada, this is done by declaring the task to be an access type and then using the allocator *new* to create an instance of the task. The execution of the *new* statement on the indicated task object is the moment of task initiation.

A single task activation is achieved in different ways according to the compiler design. In Ada, the task is viewed as a module embedded in some procedure. When program control reaches the enclosing (parent) procedure, the task is automatically activated. In PL/1, a task is formally identical to a procedure (with an attribute TASK), and it is activated by a CALL.

Thus, there are two basic concepts here: a single task activation or multiple instances of the same task.

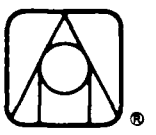
1.4.3 Intertask Communication

The third and most complex decision involves the implementation of intertask communication. This implies both data communication and transfer of control -- i.e., task invocation (call).

Data Communication: The entry of the task is equivalent to a procedure, and parameters can be passed as in a procedure. Alternatively, data can be communicated via global storage or a shared task. An example of global storage is the following program structure:

```
procedure X is
     VALUE: INTEGER;
     task A;
     task B;
     task body A is
     begin
        ...
          VALUE:=...
        ...
     end A;
     task body B is
     begin
        ...
          T:=VALUE...
        ...
     end B;
end X;
```

The latter concept of a shared task, the purpose of which is to provide common data storage, is more complex. It is similar to the Intercommunication Data Area (IDA) used in the MASCOT system [11]. A task is implemented that acts analogously to a hardware data channel. It has a read and write command and its own internal data storage. Any task may store data in it (presumably for transmittal to another task) via the write command, and can access data via the read command. Another common term for this concept is the "letterbox". There are several problems, however, that can arise. Assume that task C is the shared



task and that task A wants to transmit data to task B. Task B begins to write to C. One problem is that another task, D, may write to C, thus overwriting A's data before B reads it. A second problem relates to the way A transfers its data. If it does so in successive calls to C, then B will want to know when A has completed its transfer. A third problem is one of addressing. If A is transmitting to B, then all tasks that might read C must know that the data are intended for B. The first two problems are solved by what is called the "guard". A flag or signal is established that has two or more states; and depending upon the state value, the shared task will lock out the call from any task that could cause a problem. Ada provides for this capability by a combination of the alternative select statement (or) and the conditional select statement (when). For example, if a variable STATE can have a value "R" and "W", the task could establish a read/write guard by the construction:

```
select
     when STATE = "R" =>
          accept READ (X:in)
               ...
          end READ;
 or
     when STATE = "W" =>
          accept WRITE (X:out)
               ...
          end WRITE;
```

To prevent a second task from overwriting when in the write state, two additional variables are introduced, CURRENT_USER and USER. The former is initially "0". USER is an integer parameter that uniquely identifies the calling task. The first time it calls, it establishes CURRENT_USER. The last time it calls, it must reset CURRENT_USER. Then we would have the following guard:

```
select
      ...
or
     when STATE = "W" and CURRENT_USER = USER =>
          accept WRITE (X:out; USER in)
          end WRITE;
```

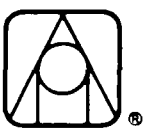

Addressing can also be implemented in a similar way, where the writing task transfers an identifying code for the intended receiver, and a guard is established on the read for this code.

Task Invocation: In describing task invocation, assume that task A calls or invokes an entry in task B. There are now four more concepts to be considered. First is synchronization. In Ada, there is only one type of inter-task synchronization; hence, no decision is necessary by the designer. This is the rendezvous or fully synchronized call. It means that task A is suspended until the entry in task B completes the processing of A's call. A point of rendezvous is established within B. If the normal processing of B arrives at the rendezvous before the call from A is reached in A, then B waits. If A calls the B entry before B arrives at the rendezvous, then A waits.

The alternative is concurrent processing. This is not available in Ada. Its implementation is discussed further in Section 3. Concurrent processing implies that task A continues to execute along with B. There are then two subcases. In the first, A must poll B to determine if it has completed service of its call. In the second, B calls A upon completion. The latter presents significant implementation problems because B must know A's identity.

The second inter-task concept and decision point is the queue type. Again, Ada permits only one queue type, FIFO. If A calls B and the task is not at the rendezvous, then A is put onto a queue. All subsequent calls to B from other tasks are also queued. In the case of concurrent processing, it is possible for A to appear multiply on the queue. In Ada, the organization of the queue is FIFO. In principle, other organizations are possible, and these are also discussed in Section 3.

The third concept is the condition under which A enters the queue. Ada presents the designer with three choices here. First is unconditioned entry.



This means that A joins the queue and remains on it until its call is serviced, however long this takes. The second option is not to join the queue at all. If the call cannot be serviced immediately (i.e., B must be waiting at the rendezvous), then it is cancelled. The third option is to join the queue for a specified time period. If service does not begin within this time, then the call is cancelled.

The fourth inter-task concept is the selection of which queue to join. In Ada, this is merely a function of which entry is called. However, a variety of structures and controls can be applied within B. First is the fixed vs. variable entry structure. In the former, one or more entries are declared, each with its own unique name. In the latter, a single entry name is declared with an index such that a particular entry is selected from the indexed series by including an index value in the call. This allows dynamic selection of the entry at run time, based upon program conditions that would assign appropriate values to the index prior to the call.

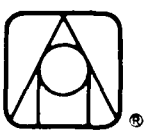
As indicated in Figure 1-5, after the fixed or variable entry decision is made, the type of entry can be selected. The fixed entry has two choices, simple or multiple. The simple entry means no alternative choices. The task has only a single entry. The multiple-entry choice represents a selection from alternative entries, as was illustrated in the last section. This is a special construction of Ada. As shown in Figure 1-5, there are three forms. First is the non-conditional alternative, or. The task waits at the select rendezvous until one of the entries is invoked. Second is the use of a logical condition, which must be satisfied before the accept. Third is the use of a delay statement in addition to the accept statement. If none of the accept statements is satisfied within the specified delay period, then control transfers to the delay block of statements, thus terminating the rendezvous. This is a way that the task can protect itself from a permanently unsatisfied rendezvous or lockout. Of course, it can return control to the select (i.e., the rendezvous) again. An even more abrupt termination of the rendezvous can be acheived by the fourth option, which is to add an else statement to the select. This is the equivalent

of a zero time delay. If no call is queued when the task arrives at the rendezvous, control transfers to the else block and the rendezvous terminates.

In summary, inter-task communication presents the designer with a number of choices and decisions: first with regard to how data are communicated and then with regard to how control is transferred. The responsiveness of the system can be critical to the latter because it determines whether or not and for how long a calling task stands in a queue or whether or not and for how long a service task waits at a rendezvous.

1.4.4 Termination

The last basic concept is task termination. Like a procedure, a task can end normally by reaching its end statement. It can also be terminated upon command. In Ada, it can terminate itself by the command terminate; it can be terminated by another task via the command abort; or it can be terminated if its parent is terminated. Termination of a task means that it ceases to run; however, there are circumstances when a task may be suspended but not terminated. That is, it is held temporarily in a non-executable state until some event occurs. These may be called "quasi-termination". One is the delay; a second is the rendezvous; and a third is the exception. The exception causes immediate transfer of control to a special block of code, with suspension of the running task. In Ada, the control of the task can be handled in two ways after the execution of the block of exception code. One is to terminate the task permanently; the other is to return control to the task that called it.



2. THE DOMAIN OF SOLUTIONS

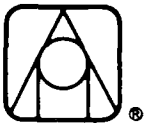
The approach toward a solution of the overall problem of embedded software development should proceed from two directions. One is a good understanding of the fundamentals and the technical details involved in each of the four problem dimensions, separately, as discussed in the Introduction. Second is to integrate the four into a single, coherent methodology.

The remainder of this section will discuss the four solution domains individually, and the next section will discuss their integration.

2.1 COMMUNICATION

Figure 1-1 shows that the essence of the problem is that four different disciplinary types must each provide its own technical input to the development process, and then must communicate it to one or more of the others. In a sense, this requires four different "languages" and either a translation or a means for one to understand that of the other. The latter approach is far more practical in the short run and possibly the long run as well, although there are those [12,13] who are working on methods for automatically translating from one stage to the next.

A significant lesson learned from the software specification studies of the NRL group [14] is that it is both feasible and highly beneficial to express requirements in a formal way. They utilize an input-output table structure, where the functions and effects of all signals in the system are rigorously specified. Also specified are erroneous conditions and unallowed states of the system. Narrative description is incorporated, where needed, but is minimized by the comprehensive use of the input-output tables. The effect of the technique is to enhance both completeness and consistency of the specification, because both visual as well as automatic checks can be made of the tables.



Furthermore, inasmuch as these tables represent a formal, computer oriented structure, the NRL work, as well as Figure 1-1, suggest a further need, which is a tight feedback between the specifier (Person A in Fig. 1-1) and a software analyst (Person B in Fig. 1). In time, it may not be unreasonable to expect that the technical training of engineers who are responsible for specification would be such as to qualify them as a Person B. This would provide the most ideal link between AB (Requirements Specification) and C (Design). Although the NRL specification and resulting documentation approach appears to this investigator to be very well suited to embedded systems, the SREM [3] method also has similar capabilities. SREM uses a different linguistic approach, but is nonetheless quite formal and achieves a similar result to the tabular approach of the NRL group. SREM defines four basic syntactic elements:

SUBJECT
ATTRIBUTE
RELATIONSHIP
OBJECT

Reference 15 presents a summary description of this method. For the purpose of this discussion, it is sufficient to say that this syntactic structure also provides the necessary input-output relations. Another, but somewhat less suited method for embedded systems, is PSL/PSA [4].

The important functions that any of these methods perform are that they must:

1. Enable the system to be organized conceptually into distinct processes.

2. Have both a formal structure to assure completeness and consistency and an informal method for incorporating natural language description, where required.

3. Enable a hierarchic breakdown of the processes in terms of specification detail.

4. Enable *control* and *data* relationships between the processes to be expressed and distinguished.

5. Enable computational algorithms to be unambiguously expressed. This will normally involve mathematical expressions and will reference the already specified inputs and outputs.

The Designer's language is normally one of charts, tables and diagrams that describe the problem specific system modules, submodule breakdowns, data structures and intermodule control structures. He/she will also become involved with the method of implementing computational algorithms. Specific languages were discussed in the previous section. However, the principal objective of the five items listed above for the specification language is to create a complete and unambiguous specification for the designer. This will fulfill the communication requirement between the AB persons and the C person.

The design process itself has an interesting internal structure, based on its purpose, which is to translate the requirement specification into a detailed (again, complete and unambiguous) design statement for the programmer (person D in Figure 1-1). It may well be that the AB-C communication link and the C-D link represent the most critical problem in the entire development process, in comparison with the internal methods employed in the AB and the C processes. In this regard, design is normally viewed in two phases: High and low level. The high level performs the essential translation from the specifications into a solution oriented form. The low level transforms the high level to the detailed and unambiguous statement required by the programmer. The previous section discussed various design documentation methods, which can fulfill either the high or the low level requirement, or both. The one that is becoming most favored, at least for low level design, is the Program Design Language (PDL) [9]. It has the following advantages:

(1) It can mix high and low level design statements.

(2) It has the same programming specificity as flow charts.

(3) It can be documented entirely with an ordinary word processor, and hence is easily used by the designer or a clerical assistant. This greatly facilitates its original generation, its maintenance, reproduction, and distribution. These mechanical expedients should not be underestimated. First, they save money by labor

reduction, even though it may be clerical or drafting labor; second, and more important, it speeds up the turnaround between designer and implementer, when design revision is required. At this point it is worth mentioning an untidy aspect of the development process. The ideal lifecycle chart, such as Figure 1-1, shows a clean break between design and implementation. The implication is that the design process can be completed and documented before the implementation begins. In principle, this ideal is attainable, but somehow in practice it rarely happens. In other engineering disciplines it is almost mandatory, because of the "concrete" nature of the product, but in software, since modification of the end product can always be performed, we frequently find some amount of feedback between the D and C persons of Figure 1-1, even as the implementation is in process. This results either from incomplete or ambiguous design documentation or from impractical or inefficient design that is detected by the programmer in the course of implementation.

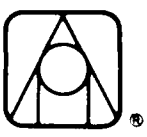
The principal document emerging from the implementation is the program. It serves the purpose of the programmer as he/she is writing and debugging the code, and of the persons who will maintain or evolve it. There is relatively little controversy over the form that this document should assume. There are simply a few "good practice" conventions that make it more readable and that can serve to better connect it to the design documentation. These are:

1. Use a consistent indentation scheme to elucidate the program structure in terms of the IFTHENELSE and the FOR nestings.

2. Use comments (generously) for three purposes. The first is to abstract each procedure. The second is to explain the use of variables in data structures. The third is to provide special explanation to any code sequence that the programmer feels may not be obvious to someone else based upon a reading of the code itself.

3. Signpost the code to the design documentation. In this way a maintenance programmer can easily cross reference between a given part of the design and the specific blocks of code that implement it.

Another possible direction that program documentation may take is to "fill in" the PDL with code. The PDL is purely textual and is sometimes referred to as pseudocode. This means that it adheres to the structure of a program (SEQUENCE, IFTHENELSE, FORWHILE), but mixes natural language directives

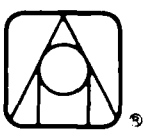


with syntactic code. (Appendix B contains an example of a very simple, but effective, PDL for the simulator developed under this contract). Therefore, in principle, the programmer could implement the program by converting the natural language pseudostatements into code, leaving the pseudocode in place to serve as the above mentioned comments. Then the program documentation would be an expanded version of the design documentation. The reason that this does not generally work is similar to the reason given above for the C-D feedback loop; namely, that the designer does not always understand how a program, or a data structure, is best structured for implementation; hence, the programmer may use the design document as a general guide rather than as a specific template for his/her program.

The communication problem is largely over after implementation. Proper specification enables one to develop the appropriate system tests. This is not to minimize the importance of this phase, but if it is the case that the specification states what the system is supposed to do, then a system test can be based entirely upon this statement. Thus, the specification document serves two purposes. It is normally AB type persons who will design and perform the system test. Ideally, it should be individuals other than the ones who specified it, but in many cases, because of economic constraint, it is some of the same persons. The designer and programmer will, of course, become involved as it becomes necessary to resolve and correct problems.

Maintenance is defined as the correction of problems after a system is in operation, but based upon the original specification. However, in time, it is often the case that it becomes necessary to change some part of the specification and to repeat, in part, the subsequent life cycle steps. In this way a system is said to "evolve". Both the specification and the design documentation are vital to this process.

In this subsection we have discussed the communication problem and the domain of solutions. Specific types of documents and methods for producing them



were also discussed. In terms of prior discussion, the focus was on method and to some extent tools, not methodology. This will continue to be the case as we examine the other problem dimensions. Methodology will emerge in Section 3 when we integrate the solution domains, and specific tools are discussed in Sections 4 and 5.

2.2 PRINCIPLES AND PRACTICES

The conventional wisdom in the past was that design methodology should be independent of the implementation language. The rationale was that the implementers should be able to choose the best language for the given design, or that it should be possible to implement the same design in several languages, possibly to run in different computer environments. This thinking has now been reversed by many, with the development of Ada [16], inasmuch as it has been mandated as the standard language for embedded systems. Also, Ada has incorporated features that encourage the use of good design principles, such as top down design, modularity, and abstraction. It also enables one to incorporate bottom up design features as well. Booch [17] says that "Ada helps break the von Neumann mind-set and lets us deal with our solutions in terms of the actual problem space." He goes on to propose what he calls *object-oriented design*:

(1) Develop a solution strategy for the problem.

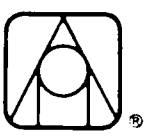
(2) Identify all of the data objects required in the solution and define their structure.

(3) Identify all operations on these objects and specify their algorithms.

(4) Establish the control and data interfaces between the operations.

(5) Implement the operations.

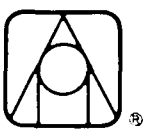
This method is in contradistinction to the various functional decomposition methods [5,8,18], wherein one first defines the functions (i.e., operations) of a system and then the data objects required by them. Furthermore, the functions are defined hierarchically. The top level functions are successively decomposed into subfunctions, until functions at the level of a compiler

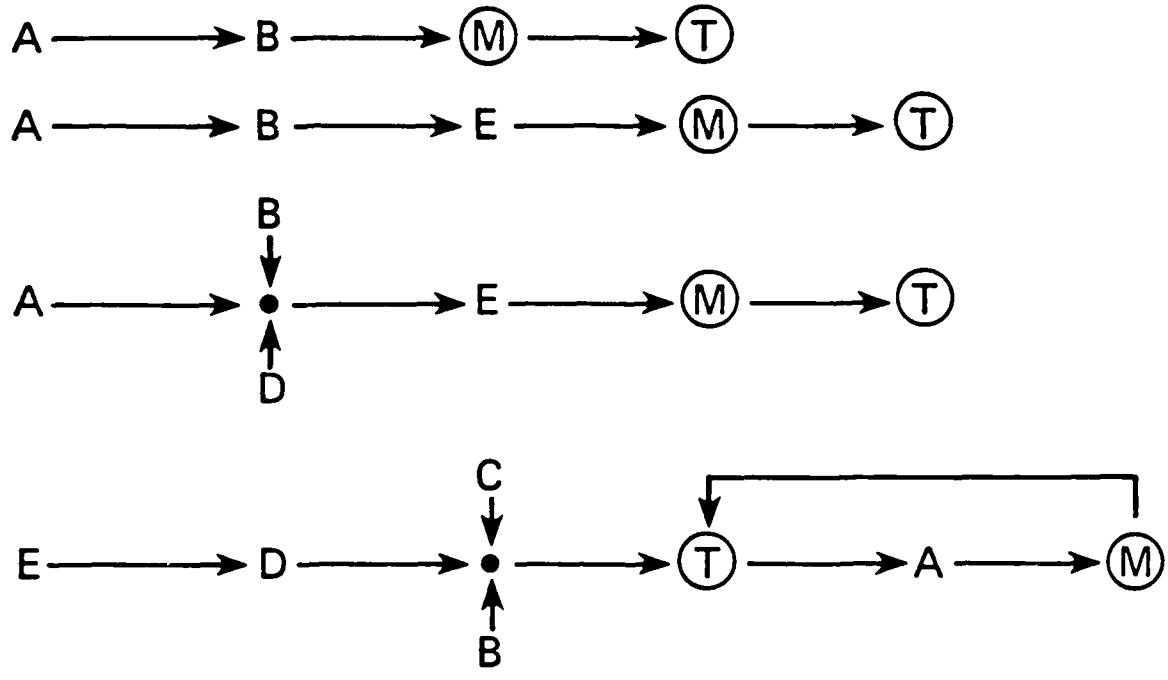


procedure are reached. In practice, both of these methods are actually used, in a mixed mode. To impose one or the other in a pure methodological form is counterproductive. Rather, a broader framework for methodology development is needed that would enable either or both of these methods to be employed by the individual practitioner.

Figure 1-2 presented the more general intellectual environment of the designer and programmer, within which are subsumed (notably A, B and C) the object oriented vs. functional decomposition methods. The issue, rather, may be one of our approach toward the integration of these five factors. Reference 15 has stated the problem as follows:

> Perhaps a methodology for embedded computer system design cannot proceed from a set of general principles. Rather, we must look at design (and specification) in terms of elemental cognitive processes. By this we mean the thought processes that the good designer must perform, at a fairly basic and low level, in order to achieve his goal. This applies both to the thoughts themselves and their organization. The issue here is not psychological --we are not interested at this point in attitude or aptitude, but rather in the basic technical issue of how a good designer breaks down the problem, from analysis to specification or from specification to design, into units of thought that we can then examine for the purpose of developing a methodology. For brevity, we call this type of examination a Cognitive Process Breakdown (CPB). Its purpose, restated, is to find the most elemental or atomic units of thinking that compose a design, so that we can then:
>
> (1) Classify these units into intellectual vs. clerical functions.
>
> (2) Organize them into logical sequences and hierarchies.
>
> (3) Identify those that may be implementation language (compiler) dependent.
>
> (4) Classify them in relation to certain high level design criteria such as time/space optimization, maintainability, reliability, and implementation speed.
>
> (5) Analyze the intellectual functions to determine whether other fields of computer science can contribute to the development of automated aids.





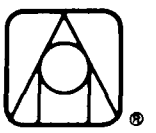
A = General Principles of System Design
B = Principles of Software Engineering
C = Traditional Practice
D = Implementation Language (Ada) Factors
E = Problem Environment
(M) = Methodology
(T) = Tools

Figure 2-1. Approaches to Methodology and Tool Development

Figure 2-1 presents some of the scenarios that could be used to integrate the five factors of Figure 1-2. The first three cases of Figure 2-1 suggest that the development of methodology progresses from the general to the specific, following a clockwise rotation around the pentagon of Figure 1-2. In the first case only high-level principles (A and B) are used to develop methodologies; they do not take into account such specifics as the implementation language or special requirements of the problem environment. The second case is somewhat more realistic in that it targets the methodology to a particular class of problem, and the third case takes into account the implementation language.

The reverse approach is represented by the fourth case. One starts with a careful examination of the specific processing and design implications of the problem class. Then the capabilities of the implementation language are considered. For example, does the multitasking capability of Ada respond to the multitasking or parallel processing requirement of embedded systems? Then one proceeds to the higher-level considerations of the existing and traditional practices vis-a-vis the newer principles of software engineering: (1) Are they in conflict? (2) Are they appropriate to the nature of the design and implementation environment as determined by E and D? At this point, one comes up not with a methodology but with a set of design and implementation-related problems that can best be solved by specific tools. Some of these problems and tools are to be discussed further in the report, but they fall into five areas: (1) design development, (2) communication from designer to implementer, (3) optimal utilization of Ada from highest to lowest level of design, (4) verification, and (5) economical but effective documentation for future maintenance.

In this context, methodology evolves from the use and refinement of these tools, taking into final consideration the more general principles of good system design. This is in contrast to the previous cases in which a methodology is first prescribed and tools are then developed to give particular form to the methodology.



The contrast in approach can also be expressed as deductive (general to specific -- clockwise around the pentagon) versus inductive (specific to general -- counterclockwise around the pentagon). Yet another expression of the latter approach is that "necessity is the mother of invention." We invent tools to solve specific problems.

The purpose here is not to say that the more conventional, deductive approach won't work, but rather to observe that we seem to have a proliferation of methodologies, yet continue to feel insecure about the prospects for producing application software in a predictably economic and reliable manner. This raises the question of whether our approach to methodology development is correct; hence a different approach is being offered.

2.3 TECHNOLOGY

There seem to be only two approaches to this problem. One is to educate or re-educate computer scientists in these technological advances. We tend to think in terms of the generations of computer hardware and languages but not that of the practitioners. This is not to say that those who were educated with second generation computers and languages can't cope with fifth generation systems, but they may require some formal updating of their skills. The second approach is less expensive and may be very effective in certain situations, but is not as broad-based as the first. This is to mask new technology through the software. For example, Ada's multitasking could be made to apply to either a multi or a distributed processor system via the operating system. Similarly, as AI techniques in rule-based languages are proven effective, they could be incorporated into commonly used languages in a semi-transparent way. That is, the users would know that rule-based capabilities can be combined with conventional language functions, but relatively little additional training would be required to use the new capabilities effectively.

2.4 COMPUTER LANGUAGE AND CAPABILITY

2.4.1 Structure of the Ada Language

If the implementation language is to be incorporated into the methodology development, then one must find those attributes of the language that relate to and re-enforce human thought processes in each of the lifecycle stages. In this particular case the language is Ada. Figures 1-3 and 1-4 present a view of Ada that may somewhat elucidate this point. Ada has four types of program unit, where a program unit is formally a separate compilation unit. For our purpose, we consider the more abstract or conceptual meaning of a program unit. Figure 1-4 shows three of these units across the top. The fourth is called a generic program unit, which, for our purpose, is conceptually similar to a subprogram. The *subprogram* would be recognized by a FORTRAN or COBOL programmer as a main or subroutine, and, in fact, is the only program unit in those languages. PL/1 programmers would recognize the *task*. It is functionally the same in both languages, though in PL/1 it is, again, formally a subroutine or procedure but is a subprogram (subroutine, procedure) that can run con-currently with others. The *package* would be recognized by the PASCAL programmer as a module, though it functions in a more general way in Ada.

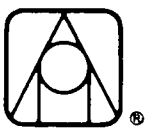
The figure shows that one high level way to conceive the program, and by extension the design, is by INTER and INTRA views of these program units. Further on we will discuss the fact that these views actually form a rather complex network, wherein one can define a set of node types corresponding to the program units and relationships among them.

The task relationships form a very fundamental part of the overall design of the system. Relationships among packages, on the other hand, are not fundamental to design, but are more of interest in the management of the programming and its subsequent maintenance. Section 2.1 distinguished between high and low level design. The inter-task view is of primary concern to the high level design, while the inter-subprogram view is of concern primarily to the low level design.

The intra-subprogram view reveals two parts: visible and hidden. The visible is that part (called the specification) of the unit that can be seen by other units that will use it. The hidden part (called the body) is not seen from outside. The visible part of a subprogram is the conventional subprogram name and the parameters that are passed to or from it. The hidden part of the subprogram are the three program components: data types, data objects, or any program unit. Below this are the procedural statements, which are of three kinds: calls to tasks, calls to other subprograms, and the normal procedural statements of assignment; if, for, I/O, etc.

A package may contain one or a combination of the three program components. These can all be made visible to users of the package. The package may also have any of these components in the hidden part as well, which is to say that they are internal to visible components, but not seen, per se, from the outside. Note that, unlike the subprogram or task, the package itself does not control the execution of procedural code. Its callable program units are called from within a subprogram or task.

A task contains executable code within it that is formally identical to a subprogram but is called an entry. The entries are the explicitly visible part of the task; however, unlike the subprogram, there is a somewhat complex structure within the task that can establish conditions for selecting the entry. These are labeled "select structure" in the figure. It establishes the alternative entries that can be called at a given point of rendezvous within the task, can impose logical conditions on the accessibility of the entry (called a "when guard"), and can impose delays that enable other actions to be taken, on a timed or periodic basis, within the task at the point of rendezvous. Though this select structure is formally in the body rather than the specification, it is nonetheless information that the designer would regard as "visible", because it affects the use of the task by other tasks. The hidden part of the task may or may not declare types, objects, or other program units. It may be purely procedural.



We are thus presented with a set of program elements that are the basic building blocks of not only the program but the design as well. They may also be applicable to requirements specifications. These can be enumerated as:

1. Subprogram
2. Package
3. Task
4. Data Types/subtypes
5. Data Objects
6. Procedural statements

Figure 1-3 indicates that they can be viewed as a network of multi-valued nodes and multivalued branches, as follows:

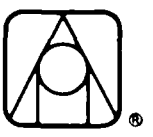
NODES

1. Types/Subtypes
2. Data Objects
3. Program Units

BRANCHES (Relations)

1. Defined/Defined by
2. Declared/Declared by
3. Used/Used by
4. Called/Called by

The design of a system could therefore start with the notion of this network of conceptual program components. Tools could be developed that would aid the designer in this process and create a documentation base that would persist throughout the remaining lifecycle stages. Moreover, this documentation base would be filled in as the development progressed through detailed design, programming, maintenance, and evolution. Section 3 will discuss the integration



of the four problem and solution domains presented above. The concept of such a documentation database will be introduced there along with some other tools that could then be formed into a coherent methodology.

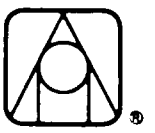
2.4.2 Limitations of Ada

Figure 1-5 indicates certain multitasking capabilities that Ada lacks but which are technologically feasible. The question then is whether these are either desirable or essential now or in the future.

2.4.2.1 Priorities. Priorities of tasks cannot be changed under program control at run time. The only dynamic priority assignment at run time is the inheritance of a calling task's priority (if higher) by a called task. This capability becomes less important in multi or distributed systems, because priority is only used as a means for deciding which task gains control of available processors. In a single processor system some advantage may be gained if the programmer has the ability to re-assign priorities, based upon conditions that are only known while the program is running. PL/1 has this capability, but it could only be added to Ada by implementing a callable procedure within the operating system that is performing the actual control of the processors and their assigned program segments.

2.4.2.2 Queue Management. Ada provides one queue ordering discipline. The ordering is always FIFO. Another possibility is to order the queue by priority. The programmer would then have to be provided with an option to assign a time at which an unserviced call would be serviced in order to prevent lock-out. Another option would be to enable a call to preempt all others on a given queue, a counterpart to the demand interrupt in hardware.

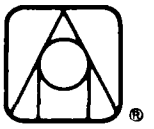
2.4.2.3 Task Concurrency. When a task is initiated it "runs" concurrently with all other initiated tasks. Of course it only is executing if there is a CPU available, and a CPU can only execute the program of one task at a time. The



CPU is therefore actually time-shared among the tasks currently in a "running" state. Contention is resolved by task priority. A multiprocessor can have more than one task in execution concurrently; however, if two tasks are in a run state, and one of them calls the other, then the calling task is suspended, even if queued, until the service is complete or cancelled. It can be cancelled only if the call had a time condition, in which case the call will be cancelled when the time given in the condition expires. This is done to simplify the synchronization of the two tasks. The called task will accept the call when it arrives at a point of rendezvous within the body of the called task. At this point, as discussed above, there may be a select structure that enables alternative entries (only one of which is intended to respond to a particular call), and some of the entries may have a "when guard"; if the guard is closed, then the call will not be accepted (i.e., it remains queued) until the guard opens.

Although one cannot add other queue management disciplines or modes of synchronization directly to Ada, it is possible to add these capabilities indirectly by adapting an idea that was used by Kiviat and Pritsker [21] in their GASP simulator in 1969 and by Simpson and Jackson [11] in their MASCOT "machine".

Kiviat and Pritsker used it to implement a discrete event simulator in FORTRAN. Simpson and Jackson used it to add inter-task communication facilities to a modified version of Pascal, called PALE (Pascal Language Extension). Since the technique has neither been uniquely identified nor named in the literature, we will call it here a Virtual Real-Time Machine (VRM). Figure 2-2 illustrates the method. The VRM is a set of procedures written in the language of the compiler. Its purpose is to enhance the apparent capability of the compiler by providing an additional set of commands to the application programmer (AP). Of course, the enhancement is only a virtual or apparent one because the AP could have written them himself directly. These commands form a coherent subsystem that causes the total package (compiler and VRM) to look like a different compiler or "virtual machine". In the case of Kiviat and Pritsker, they transformed FORTRAN II into a parallel processor by enabling multiple events to be defined that had



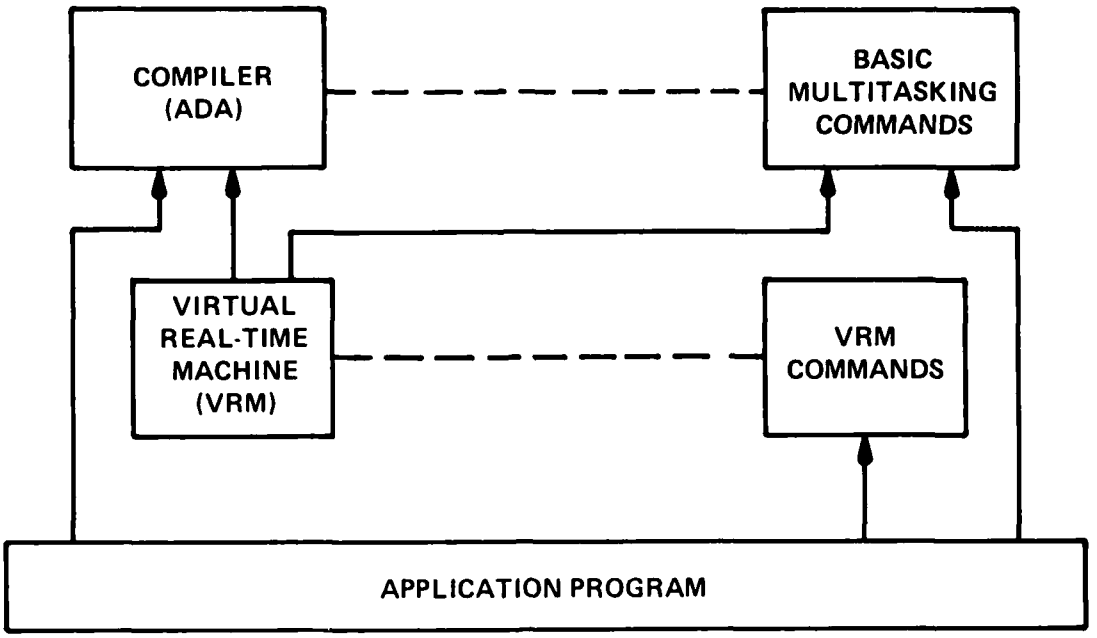


Figure 2-2. Relationship of the VRM to the Compiler and the Application Program

the apparency of running concurrently. In their case, the compiler didn't even have any basic multitasking commands. They simulated them within GASP (their VRM). In the case of Simpson and Jackson's MASCOT, they used the task scheduling facility of PALE (which was its essential Pascal extension), but added subroutines to establish independent tasks called Intercommunication Data Areas (IDAs) to control the transfer of data and the queueing between tasks that needed to communicate. MASCOT need not be introduced here, since it is more relevant to the VRM. Its construction will therefore, be explained further.

Figure 2-3 shows that MASCOT imposes a particular structure on the application program, which implies a particular method of system design. The application tasks (i.e., of the target embedded system) are all defined as activities and are written in the language of the compiler. Activities interface to other activities through software components (themselves tasks) called IDAs. There are two types of IDA: a channel and a pool. A channel is both a synchronizer and a data buffer between two activities. As such, it has an input or entry from a provider task (activity) and another input or entry

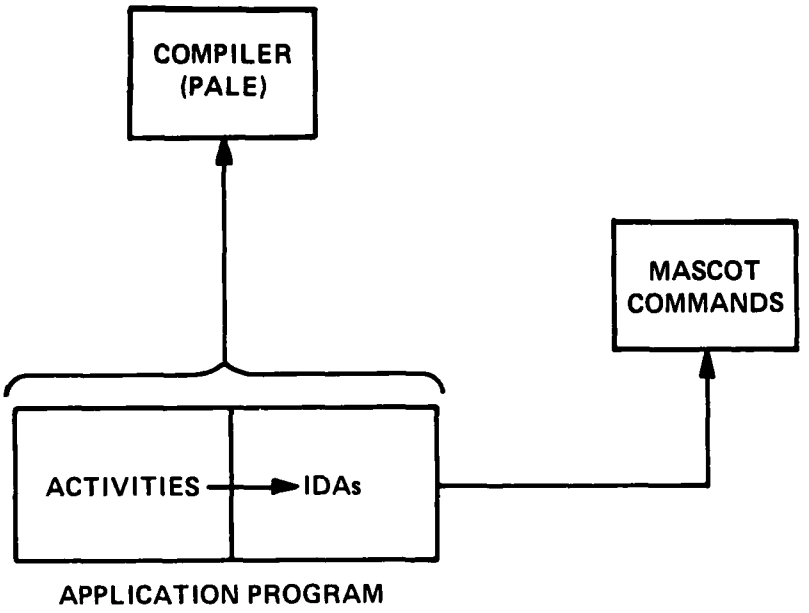


Figure 2-3. MASCOT System Design Methodology

from a consumer task. Schematically, it is represented as shown in Figure 2-4, asymmetric in that A calls B. In actuality, the implementation is as shown in Figure 2-5. The solid line indicates control and the dashed line indicates data transfer. A actually calls I in a fully synchronized mode (i.e., A is suspended until I completes its service, which is usually a simple buffering of data from A). B also calls I to receive the data placed there by A. I synchronizes the two so that even if B's call precedes A's, B will be suspended by I until the A call arrives and is serviced. This is an example of symmetric synchronization, because B really must know about A inasmuch as it must specifically call I for the data that A will place there. The effect is to release A faster than might be the case if A had been queued before B. However, if A was expecting a response from B, then either I would hold A until the response came back or A would call I again for the response. This type of approach is sometimes referred to as the "letterbox" approach, where tasks communicate through an intermediate task that buffers back and forth.

A MASCOT pool is a somewhat more complex type of intermediary which is more like a file storage system. Tasks can store and update in it, and other tasks can share data in these "files". Figure 2-6 illustrates the pool schematic.

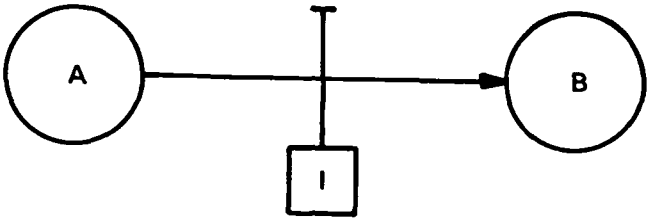


Figure 2-4. Schematic of a Channel I

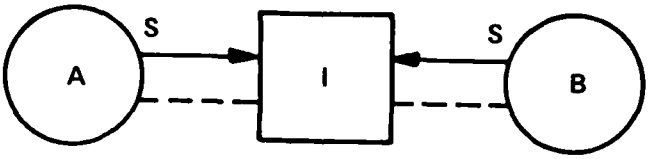


Figure 2-5. Implementation of a Channel I

Appendix A contains a specification and a design concept for a VRM adjunct to Ada.

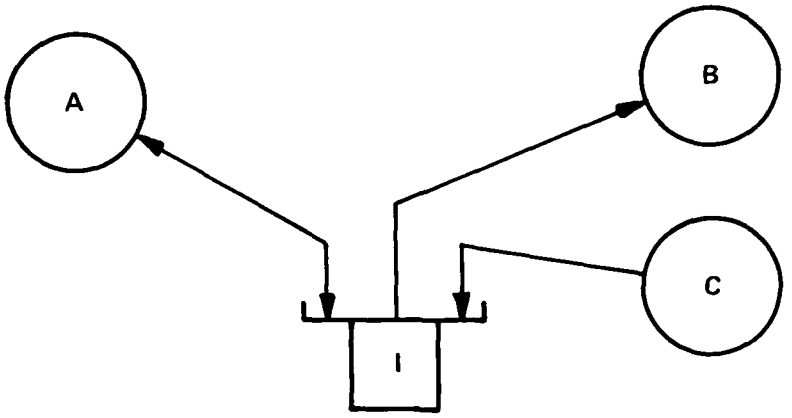


Figure 2-6. *Schematic of a Pool I*

3. INTEGRATION OF THE FOUR SOLUTION DOMAINS

This research has characterized the problems of developing a methodology for embedded system development as occuring in four distinct domains:

(1) Communication among and between four different technical disciplines.

(2) Current principles and practices in software engineering.

(3) Technology of software and hardware.

(4) The specific structure and capabilities of the Ada compiler.

Section 1 discussed the nature of each of these problems, and Section 2 discussed solution approaches within each. It was also proposed that a development methodology should integrate these individual solution domains. This can be done by means of the following set of concepts and techniques:

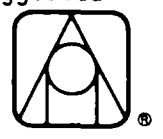
(1) A set of documents and tools is identified (and developed) for each lifecycle stage.

(2) The documents are established in a common database. A special purpose Database Management System will provide for the maintenance of these documents and for oversight of the status of the entire lifecycle by a system manager.

(3) The concept of iterative development will be introduced. This concept embraces the idea that there is a more fluid back and forth flow between lifecycle stages and that the form of documents, the tools, the DBMS and the system management should all take this into account.

3.1 THE CONCEPT OF ITERATIVE DEVELOPMENT

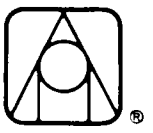
This concept is based upon two premises. One is that no matter how we have tried in the past to develop a one-directional path of system development stages, in practice we always seem to retrace, because (1) a former stage is incomplete, erroneous or ambiguous, (2) improvements are subsequently suggested



and accepted, necessitating earlier stage changes, or (3) requirements change. The NRL work of Reference 14 is important in this regard. It can be seen as an attempt to state complete and unambiguous requirements, so that retrace to this stage is never required, unless the requirements change. To the extent that this kind of documentation activity can be successful, in each stage, the concept of iterative development becomes less valid; however, this is an ideal, and it may be just as well to accept some degree of looseness or imperfection in these documentation processes, compensating for them by the creation of a communication process and an overall methodology that admits this weakness by formalizing the feedback process. Stated in another way, we may think of the five personnel types A,B,C,D,E of Figure 1-1 to be a team whose efforts are coordinated through specific documents, with smooth interfaces, mechanized via the DBMS.

The second premise is that the Ada component network, discussed in Section 2.4, should become the conceptual framework for the system design and possibly even its specification. For example, the NRL specification tables can be considered to be program units and the parameters within these tables to be data objects. Typing of the objects might in some instances be done in the specification; in others it would be filled in at the design stage. Implementation detail of the program units would, of course, also be provided in the design stage. That is, design of algorithms as procedures, identification and composition of tasks, specification of inter-task communication, specification of package compositions, design of complex data structures, etc.

As for whether these three components: Types, Objects and Program Units, are manipulated via the rules of functional decomposition (program units first, then objects and types) or via the object-oriented design (objects and types first, then program units), can be left as a matter for individual project management decision. The important thing is that the result is conceptually and then in actuality the Ada component network described in Section 2.4 above. The tools, documentation and database will all be developed to accommodate this view of the system, starting as far back in the lifecycle as is reasonable.



Section 3.2 will discuss the database, and Sections 4 to 6 will discuss the tools.

3.2 ITERATIVE DEVELOPMENT AND THE DATABASE

Figure 3-1 presents a schematic of the database. Each of the layered bands represents a distinct set of files in the database that supports a particular lifecycle stage or part of a stage.

3.2.1 Requirements Specification

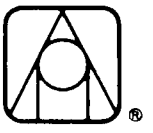
At the top is the Requirments Specification. Either SREM or some specification language that might emerge from the NRL work would appear to this investigator to be the best suited for the purpose at this time. The database representation of Figure 3-1 and the associated discussion are conceptual, as far as the DBMS is concerned. The question of whether or not and how to interface SREM or any other system is beyond the scope of this project, as is the detailed design of the DBMS. Only the feasibility of doing so is assumed here.

3.2.2 Design -- High Level

The design stage occupies three layers of the diagram. First is the high level design. At this point a tool is introduced called the Program Architectural Diagram (PAD). This is a combination of graphical and tabular information that describes the Ada component network, with its various INTER and INTRA views and at different levels of detail. It can be started either by the designer in the high level stage or by the specifier in the previous stage and completed at the low level stage. The PADs are described in Section 5.

3.2.3 Design -- Modeling

One of the PADs of special interest provides the means for precisely specifying the inter and intra task views. This is of central importance in embedded systems, because it is the parallel nature of the processing and the criticality of response time that distinguishes this type of system from other



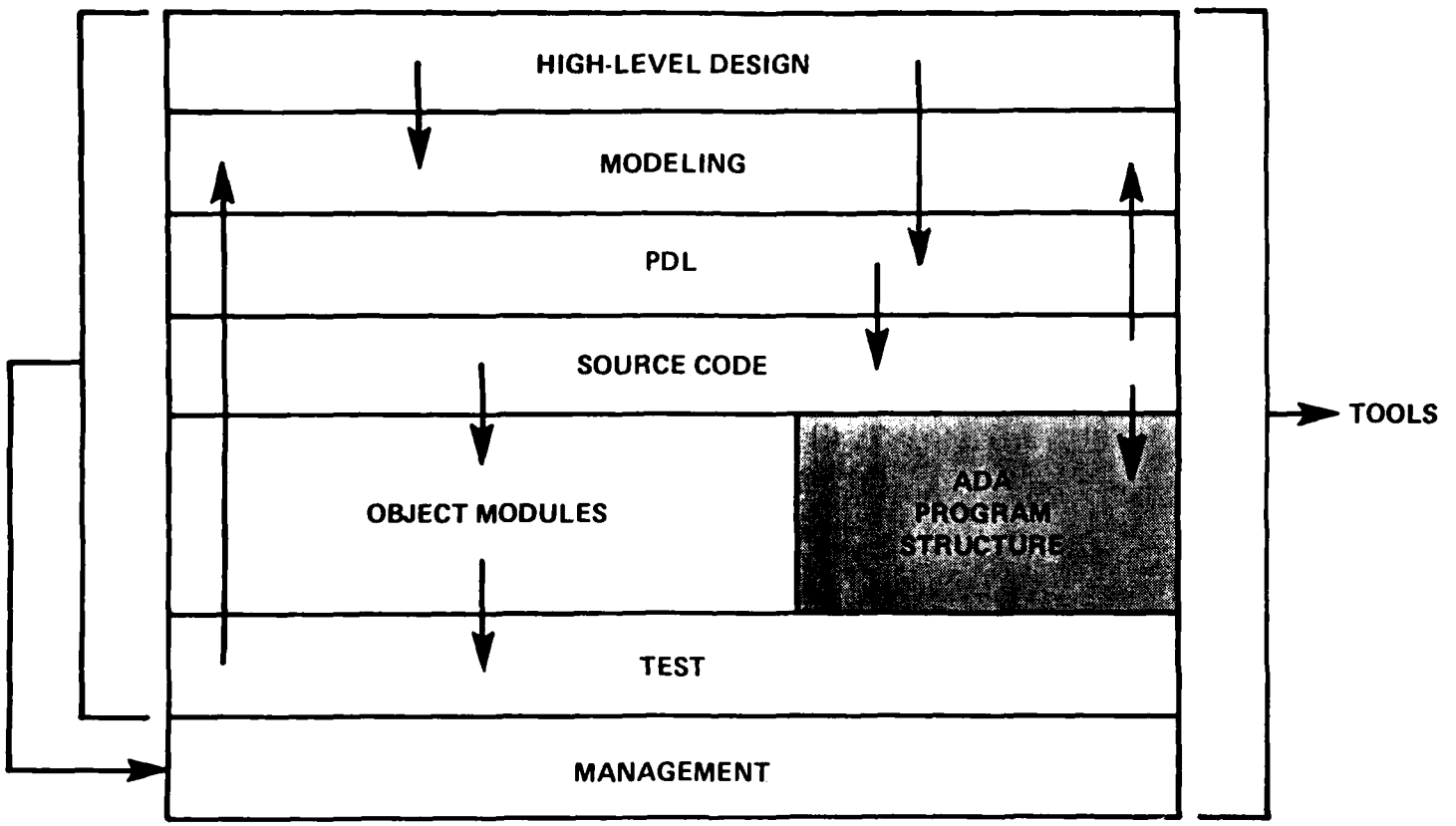
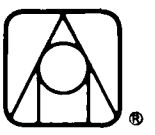


Figure 3-1. Software Development Data Base

data processing systems. Algorithms and data structures are common to all systems. They only vary in their specification. The superimposition of parallel processing is what complicates these systems. Furthermore, it is difficult to predict how a particular multitasking design solution will perform. Therefore, a database layer and tool, called modeling, appears in Figure 3-1 between the high and low level designs. If this process could be performed accurately and with reasonable effort on the part of the designer, it would be a valuable addition to the methodology. The modeling is implemented by means of a simulator and serves a number of functions, as indicated by the arrows in the figure:

1. (High-Level Design to Modeling) The model presented to the simulator consists of the tasks and their intercommunication, a breakdown of each task into internal "events", and the desired execution time of these certain events. The events are all related to Ada program constructs. The simulator is called the Embedded Software Design Simulator (ESDS). It has been developed under this project and is described in several reports and a paper [1,2,19,20]. For the purpose of this report, it is reviewed in Section 6. The result of running the ESDS is to determine whether required input-output response times can be achieved. The ESDS run report will pinpoint bottlenecks and provide a variety of quantitative performance measures related to the tasks and events. These can indicate how elastic the system is if the response requirements became more stringent or if the inputs appeared with greater frequency. The designer can then respond to this information by the following kinds of design changes: change in task priorities; change in the composition of task entries; change in the timing conditions on task calls; and change in the required execution times of critical events. Information in the ESDS run report is explicit enough to enable the designer to decide which of these remedies may be appropriate. Another more radical remedy would be to increase the number or speed of processors, which, of course, has hardware implications.

2. (Ada Program Structure to Modeling) The Ada Program Structure (APS) *is* the Ada component network. In terms of the database, it is automatically determined from the source code (arrow from Source Code to the APS). The creation of the APS file enables the system to then automatically create the PADs resulting from the actual Ada source code and to also create a model in reverse, as indicated by the arrow from the APS to the Modeling layer. The creation of the PADs is a fairly straightforward task, but the creation of the reverse model is not, although it is algorithmic. The reverse model can then be run through the ESDS in the same



manner as the previously described "forward" model. This can serve two useful purposes. One is to test the program very rapidly under a large number of stimulus conditions. If performance for any particular set of stimuli is unsatisfactory, then the run would be validated by running the actual program with these same stimuli. If correct, then the ESDS can be used, as described above, to modify the design. This is an example of iterative development.

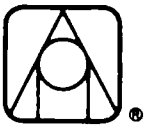
The second purpose is to compare the reverse and the original forward models of the design. Such a comparison is possible because the ESDS report is quantitative with respect to all of the important comparison factors, including each input-output response time, a histogram for each task entry queue, and the series of task and event quantitative performance measures.

3. (Test to Modeling) In this feedback path the program implementation is tested and the results compared with those originally obtained (or obtained from the reverse model). Assuming that the tests are properly performed, this is more a validation of the model than of the programs, because the success or failure of the tests serves to validate (or not) the programs, regardless of what the simulator indicates; however, once one has gained some confidence in the model, it can be used to expedite testing by enabling a large set of tests to be performed (on the model) more rapidly and at less cost. The negative test results on the model would then prompt live tests on the programs.

3.2.4 Design -- Low Level

As discussed in Section 2.1, the PDL is emerging as a good solution to the problem of documenting detailed software design. It has the advantages of accurately representing structured program designs, good readability, accommodation of unconstrained and unlimited natural language explanatory text without sacrifice of the structured format, and easy maintenance on a common word processor (no need for graphics). In addition, a formalized set of keyword directives is normally built into the PDL that encourage the user to include, routinely, important design information that is very helpful in program maintenance. As an example, Byron [9] uses the following Keywords:

Keyword	Meaning
Algorithm	General procedure used to solve the problem. What Booch would call the strategy.



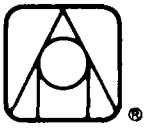
Keyword	Meaning
Effects	Function performed by a subprogram.
Errors	Error messages and circumstances.
Invariant	Properties that do not change over time.
Modifies	Global variables that may be affected.
Notes	Host dependencies or limitations.
Overview	Relation of subprogram to environment.
Raises	Exceptions that may be raised.
Requires	Assumptions by subprogram about available data in globals or parameters.
Tuning	Relationship between values and performance.

The PDL could then generate reports based upon these keywords that would cross-reference all subprograms, selectively, to each of the keyword categories. Thus, an Errors Report broken down by subprogram could be produced.

The development database would therefore incorporate the PDL and be capable of generating the various reports of the PDL.

3.2.5 Source Code and Object Modules

The Source Code is generated manually from the PDL. The programmer may also use the results of modeling as a guide to how tight his/her code must be within various subprograms. The source code, with embedded comments and signposts to the PDL, would be included in the database. From it, the object modules would be compiled and a configuration manager would be used to control the generation of load modules for debugging and system testing.



3.2.6 Ada Program Structure

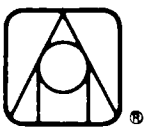
The database will contain a representation of what was referred to above as the Ada Program Structure. These are the various INTER and INTRA views of the program and, consequently, the system. They would be generated automatically from the source code. They would be displayed in a combination of graphic and tabular forms. These are described in some detail in Section 5.

3.2.7 Test

System test designs and results would be stored in the database. As mentioned, the model simulation may be used in conjunction to permit the test designer to focus on critical areas to test more thoroughly, or to help in the sequencing of the tests.

3.2.8 Management

The database has three main purposes: the first is to provide documentation methods and tools to facilitate the work within each lifecycle stage; the second is to provide a smooth communication link between stages; and the third is to enable overall management of the entire process. This can incorporate traditional charting and time/effort accounting methods, relating to personnel assignment and schedules. It can also now incorporate control over the scheduling, generation, review, and ongoing maintenance of the documentation products of the system itself. In the past, it is this aspect of management that has eluded solution and has been a significant cause of delay and cost overrun.

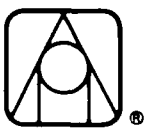


4. TOOLS

As Figure 3-1 schematically indicates, each of the database layers feeds into some kind of tool or set of tools. Also, as indicated in Section 2.2, it may be the tools that are most fundamental to the methodology, because it is these that will reflect the most practical needs and produce the best methods for performing the various functions of each stage. A number of such tools have been cited as potentially useful and compatible with the ideas of iterative development and the development database. Some are existing products and others have been specified and to some extent developed under this project to fill a perceived gap. These tools or classes of tools are listed below, with an asterisk after the new ones introduced by this project.

1. Requirements specification languages such as SREM, PSL/PSA, and the NRL method.
2. High level design languages such as SADT and Structured Design.
3. Modeling languages such as POD and ESDS (*).
4. A low level design language such as the PDL.
5. Source code and configuration managers such as Librarian and Panvalet.
6. The Program Architectural Design generator (PAD)(*).
7. A system test management system.
8. Project management systems (PERT, GANTT, etc.)

The PAD and the ESDS are described in Sections 5 and 6.



5. PROGRAM ARCHITECTURAL DESIGNS (PAD)

The PAD is based on the concept presented in Section 2.4 that a system design based upon the inherent constructs of Ada can be viewed as a network with five node types and four bidirectional branch types. The five node types are (1) Types/Subtypes, (2) Data Objects, (3) Tasks, (4) Packages, and (5) Subprograms. A set of files can therefore be established with record structures that contain all of the information to characterize every element in a given design and its consequent program within these five node types. These files comprise a subdatabase of the total development database of Figure 3-1. In particular, it is the one in the shaded region labeled Ada Program Structure.

5.1 FILE STRUCTURE FOR THE NETWORK

There is one record type for each node type in the network. Built into the record are the four network branch types, as required. Tables 5-1 through 5-5 present the possible content of these records.

Each subprogram record is assigned a unique accession number, AN. This is used as a link so that one record in the network can reference another.

5.2 NETWORK TRANSITIONS VIA THE DATABASE

Table 5-6 presents the way in which transitions are made from one node in the network to another. It is in the form of a matrix. The first section of the table shows the three transitions from SUBPROGRAMs to SUBPROGRAMs, PACKAGEs and TASKs via the relations "What do I declare", What declares me", etc. The second section of the table shows the transitions from PACKAGEs to the other three program units, and the third section shows the three transitions from TASKs to the other three program units. The fourth through sixth sections of

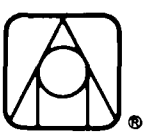


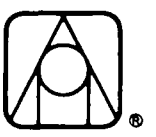
Table 5-1. Type Record

1. Type/Subtype Name

2. Class
   - (1) Scalar - non enumeration
   - (2) Scalar - enumeration
   - (3) Array
   - (4) Record
   - (5) Access
   - (6) Task

3. Subtype
   - 3.1 Indicator
     - (1) S = Subtype
     - (2) D = Derived
     - (3) X = Non Subtype
   - 3.2 Type Name

4. Type (Subtype) Description by Class
   - 4.1 Scalar - non enumeration
     - 4.1.1 Primitive representation (integer, float, string, range, etc.)
   - 4.2 Scalar - enumeration
     - 4.2.1 Values
   - 4.3 Array
     - 4.3.1 Type name or primary rep of array elements
     - 4.3.2 No. of dimensions
     - 4.3.3 Component description (R)
       - 4.3.3.1 Scalar type name or prim rep
       - 4.3.3.2 Range
   - 4.4 Record
     - 4.4.1 Data element (R)
       - 4.4.1.1 Data element name
       - 4.4.1.2 Type specification items 1 to 7
     - 4.4.2 Discriminant (R)
       - 4.4.2.1 Discriminant name
       - 4.4.2.2 Discriminant value (R)
       - 4.4.2.2.1 Data element (R)
       - 4.4.2.2.1.1 Data element name
       - 4.4.2.2.1.2 Type specification items 1 to 7
   - 4.5 Access
     - 4.5.1 Type name of the template
     - 4.5.2 Template AN
   - 4.6 Task (R)
     - 4.6.1 Entry name
     - 4.6.2 Interrupt Indicator
       - (1) I = Interrupt
       - (2) X = Non Interrupt



5. Private Type
   5.1 Indicator
      (1) L = Limited
      (2) X = Simple private type
   5.2 Operation (Procedure or function) (R)
      5.2.1 Name
      5.2.2 AN of the procedure or function

6. Program unit in which type is defined (R)
   6.1 Program unit name
   6.2 AN
   6.3 Program unit Class
      (1) S = Subprogram
      (2) P = Package
      (3) T = Task

7. Other program units in which type is used (R)
   7.1 Program unit name
   7.2 AN
   7.3 Program unit class
      (See 6.3)

Table 5-2. Object Record

1. Object name
2. Class
   (1) V = Variable
   (2) C = Constant
3. Type
   3.1 Type name or prim rep
   3.2 AN of program unit in which type is defined
4. Constraint
5. Initial value
6. Program unit in which object is declared
   6.1 Program unit name
   6.2 AN
   6.3 Program unit Class
      (1) S = Subprogram
      (2) P = Package
      (3) T = Task
7. Other program units in which used (R)
   7.1 Program unit name
   7.2 AN
   7.3 Program unit Class
      (See 6.3)

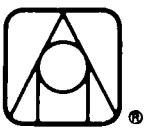


Table 5-3. Subprogram Record

1. Subprogram or entry name
2. AN
3. Parent AN
4. Subprogram/entry Ind
   (1) P = Procedure
   (2) F = Function
   (3) E = Entry
5. Subprogram/entry specification
   - 5.1 Parameter (R)
     - 5.1.1 Parameter Name
     - 5.1.2 Mode (in, out, in out)
     - 5.1.3 Type Name
     - 5.1.4 Default value
6. Subprogram/entry body
   - 6.1 Subprogram (R)
     - 6.1.1 Subprogram name
     - 6.1.2 AN
   - 6.2 Package (R)
     - 6.2.1 Package name
     - 6.2.2 AN
   - 6.3 Task (R)
     - 6.3.1 Task name
     - 6.3.2 AN
   - 6.4 Calls
     - 6.4.1 Called subprogram (R)
       - 6.4.1.1 Subprogram name
       - 6.4.1.2 AN
     - 6.4.2 Called entry (R)
       - 6.4.2.1 Task name
       - 6.4.2.2 Entry name
       - 6.4.2.3 Entry AN
       - 6.4.2.4 Conditional call Ind
         - (1) T = Timed entry call
         - (2) I = Immediate acceptance
         - (3) X = Unconditional call

Table 5-4. Package Record

1. Package name
2. AN
3. Parent AN
4. Package specification
    - 4.1 Subprogram name (R)
    - 4.2 Package name (R)
    - 4.3 Task name (R)
    - 4.4 Type name (R)
    - 4.5 Subtype name (R)
    - 4.6 Object name (R)
5. Package body
    - 5.1 Visible Program Units
        - 5.1.1 Subprogram (R)
            - 5.1.1.1 Subprogram name
            - 5.1.1.2 AN
        - 5.1.2 Package (R)
            - 5.1.2.1 Package name
            - 5.1.2.2 AN
        - 5.1.3 Task (R)
            - 5.1.3.1 Task name
            - 5.1.3.2 AN
    - 5.2 Hidden Program Units
        - 5.2.1 Subprogram (R)
            - 5.2.1.1 Subprogram name
            - 5.2.1.2 AN
        - 5.2.2 Package (R)
            - 5.2.2.1 Package name
            - 5.2.2.2 AN
        - 5.2.3 Task (R)
            - 5.2.3.1 Task name
            - 5.2.3.2 AN
    - 5.3 Calls
        - 5.3.1 Called subprogram (R)
            - 5.3.1.1 Subprogram name
            - 5.3.1.2 AN
        - 5.3.2 Called entry (R)
            - 5.3.2.1 Task name
            - 5.3.2.2 Entry name
            - 5.3.2.3 AN
            - 5.3.2.4 Conditional call Ind
                - (1) T = Timed entry call
                - (2) I = Immediate acceptance
                - (3) X = Unconditional call

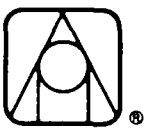


Table 5-5. Task Record

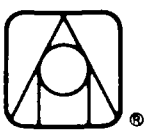
1. Task name
2. AN
3. Parent AN
4. Object Ind
   (1) T = Task type
   (2) X = Non task type
5. Task specification
   5.1 Entry (R)
      5.1.1 Entry name
      5.1.2 Interrupt Ind
         (1) I = Interrupt
         (2) X = Non interrupt (called entry)
      5.1.3 Parameters (R)
         5.1.3.1 Parameter name
         5.1.3.2 Mode Ind
            (1) I = in
            (2) O = out
            (3) B = in out
6. Task body
   6.1 Rendezvous (R)
      6.1.1 Select Ind
         (1) S = Select
         (2) X = Non select
      6.1.2 Entry name (R)
      6.1.3 Alternative Ind
         (1) D = Delay
         (2) E = Else
         (3) T = Terminate
   6.2 Subprogram (R)
      6.2.1 Subprogram name
      6.2.2 AN
   6.3 Package (R)
      6.3.1 Package name
      6.3.2 AN
   6.4 Task (R)
      6.4.1 Task name
      6.4.2 AN
   6.5 Calls
      6.5.1 Called subprogram (R)
         6.5.1.1 Subprogram name
         6.5.1.2 AN
      6.5.2 Called entry (R)
         6.5.2.1 Task name
         6.5.2.2 Entry name
         6.5.2.3 AN
         6.5.2.4 Conditional Call Ind
            (1) T = Timed entry call
            (2) I = Immediate acceptance
            (3) X = Unconditional call

Table 5-6. Network Transitions in the AdaDatabase

	From SUBPROGRAM To		
	SUBPROGRAM	PACKAGE	TASK
What do I declare:	6.1(D)	6.2(D)	6.3(D)
What declares me:	1(I-6.1)	1(I-5.1.1)Vis 1(I-5.2.1)Hid	1(I-6.2)
What do I call:	6.4.1(D)	---	6.4.2(D)
What calls me:	1(I-6.4.1)	1(I-5.3.1)	1(I-6.5.1)

	From PACKAGE To		
	SUBPROGRAM	PACKAGE	TASK
What do I decalare:	5.1.1(D)Vis 5.2.1(D)Hid	5.1.2(D)Vis 5.2.2(D)Hid	5.1.3(D)Vis 5.2.3(D)Hid
What uses me:	1(I-6.2)	1(I-5.1.2)Vis 1(I-5.2.2)Hid	1(I-6.3)
What do I call:	5.3.1(D)	---	5.3.2(D)
What calls me:	---	---	--

	From TASK To		
	SUBPROGRAM	PACKAGE	TASK
What do I declare:	6.2(D)	6.3(D)	6.4(D)
What declares me:	1(I-6.3)	1(I-5.1.3)Vis 1(I-5.2.3)Hid	1(I-6.4)
What do I call:	6.5.1(D)	---	6.5.2(D)
What calls me:	1(I-6.4.2)	1(I-5.3.2)	1(I-6.5.2)



	From TYPE to	
	OBJECT	PROGRAM UNIT
Where defined:	---	6(D)
Where used:	1(I-3)	7(D)

	From OBJECT To	
	PROGRAM UNIT	TYPE
Where declared:	6(D)	
Where used:	7(D)	
What		3.1(D)

	From PROGRAM UNIT To	
	TYPE	OBJECT
Which do I define/declare:	1(I-6)	1(I-6)
Which do I use:	1(I-7)	1(I-7)

Examples in the use of Table 5-6.

Example 1: To go from a SUBPROGRAM to all of the TASKS declared within it.

(1) Use the first section (From SUBPROGRAM To) of Table 5-6, and find the matrix entry for the TASK column and "What do I declare". The matrix entry is 6.3(D).

(2) Use the Subprogram record (Table 5-3), and look at Section 6.3. The D means "direct". That is, The set of tasks declared by this subprogram are given directly in Section 6.3. The "(R)" means that 6.3 is a repeating field -- i.e., it repeats for all tasks.

Example 2: To go from a PACKAGE to all SUBPROGRAMs that use it.

(1) Use the second section (From PACKAGE To) of Table 5-6. The matrix entry for the SUBPROGRAM column and the "What uses me" row is 1(I-6.2).

(2) This means use the Package Record field 1, which is the name of the package, as a key in field 6.2 of the Subprogram Record. The "I" means an indirect, or what database systems refer to as "inverted" search. In this case it would find

all subprogram records with the required package name in field 6.2 of the subprogram record.

the table show the transitions from TYPE, OBJECT, and PROGRAM UNIT to other TYPEs, OBJECTs, and PROGRAM UNITs.

Examples of how to use the table are given below.

5.3 DISPLAY OF THE PADs

The database records defined in Section 5.1 and the transition chart of Section 5.2 provide the means for display of the various PADs, which can be presented as a combination of tabular and diagrammatic information. Seven such displays are presented below. In the case of the diagrams, the program should be able to display the entire network globally without the detail of information at each node, or to focus on the detail of specific nodes within it.

5.3.1 Intertask PAD

1. Header
   - 1.1 Task Name
   - 1.2 Priority
   - 1.3 Object Ind
2. Entry (R)
   - 2.1 Entry name
   - 2.2 Interrupt Ind
   - 2.3 Calling Task (R)
     - 2.3.1 Task name
     - 2.3.2 Argument (R)
       - 2.3.2.1 Argument name
       - 2.3.2.2 Type
       - 2.3.2.3 Mode

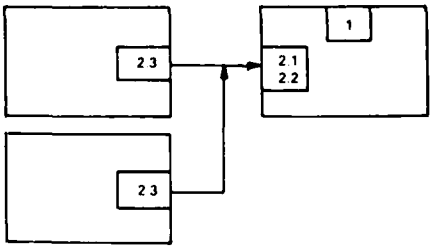


Figure 5-1. Intertask PAD

5.3.2 Intratask PAD

1. Header
   - 1.1 Task name
   - 1.2 Priority
   - 1.3 Object Ind
2. Input select structure (R)
   - 2.1 Select Ind
   - 2.2 Entry (R)
     - 2.2.1 Entry name
     - 2.2.2 When guard Ind
     - 2.2.3 Interrupt Ind
     - 2.2.4 Parameter (R)
       - 2.2.4.1 Parameter name
       - 2.2.4.2 Type
       - 2.2.4.3 Mode
   - 2.3 Alternative Ind
3. Subprogram name (R)
4. Package name (R)
5. Task name (R)
6. Called Task (R)
   - 6.1 Task name
   - 6.2 Entry name
   - 6.3 Conditional call Ind

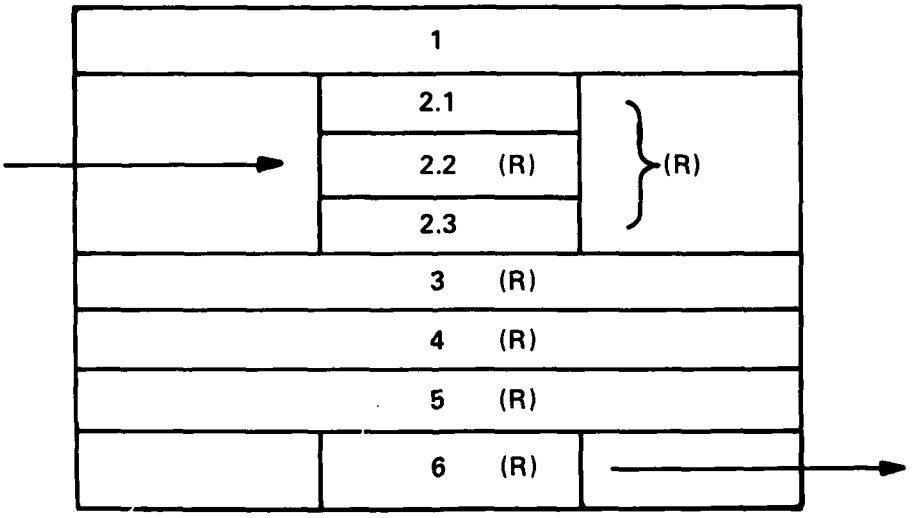


Figure 5-2. Intratask PAD

5.3.3 Inter and Intra Package PAD

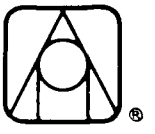
The inter and intra package PADs are purely tabular.

1. Package name
2. Used package name (R)
3. Used by package name (R)

1. Package name
2. Visible part
   - 2.1 Subprogram name (R)
   - 2.2 Package name (R)
   - 2.3 Task name (R)
   - 2.4 Type/Subtype name (R)
   - 2.5 Object name (R)
3. Hidden Program Units
   - 3.1 Subprogram name (R)
   - 3.2 Package name (R)
   - 3.3 Task name (R)

5.3.4 Intersubprogram PAD

This is the conventional call tree. It shows which subprograms call which others; however, in addition it also shows calls to tasks within the same tree structure, but these are clearly differentiated from normal intratask subprogram calls; hence, some of the branches represent parallel activity (calls to tasks) and others sequential (calls to subprograms).



There is no intrasubprogram PAD here. This would be represented either by a conventional flowcharting technique or, better, by the PDL. Each subprogram record in the database would be keyed to a corresponding PDL record, and, when the intrasubprogram detail is required, the appropriate PDL could then be displayed.

1. Subprogram name
2. Subprogram Ind
3. Called subprogram (R)
    - 3.1 Subprogram name
    - 3.2 Argument (R)
        - 3.2.1 Argument name
        - 3.2.2 Type
        - 3.2.3 Mode
4. Called entry (R)
    - 4.1 Task name
    - 4.2 Entry name
    - 4.3 Argument (R)
        - 4.3.1 Argument name
        - 4.3.2 Type
        - 4.3.3 Mode

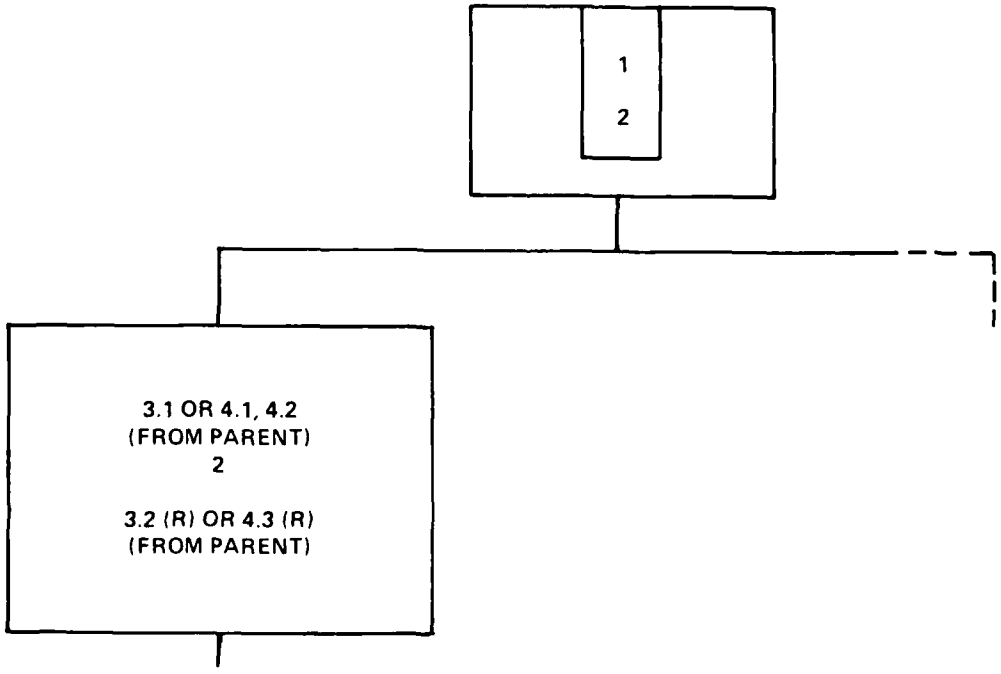


Figure 5-5. Intersubprogram PAD

5.3.5 Type and Data Object Reports

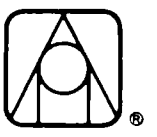
There are two reports, related to types and data objects, that cross reference these to all of the places that declare and use them. The figures simply list the column headings of each report.

1. Name
2. Class
3. Subtype Indicator
4. Attribute Description
5. Private Description
6. Unit Where Defined
   - 6.1 AN
   - 6.2 Name
   - 6.3 Class
7. Unit Where Used
   - 7.1 AN
   - 7.2 Name
   - 7.3 Class

1. Name
2. Class
3. Type
4. Where Declared (AN of Unit)
5. Where Used (AN of Unit)
6. Constraint
7. Initial Value

5.4 DISPLAY OF THE REPORTS

A particular PAD may either be displayed in its entirety on paper or can be displayed selectively on the screen, where it would normally be all information associated with a particular subprogram, task, package, type, or object. The display system would also have the ability to show the global network or to zoom in on any node or set of nodes in detail.

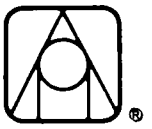


6. THE EMBEDDED SOFTWARE DESIGN SIMULATOR (ESDS)

One of the tools discussed in Section 3 was a simulator for high level designs of embedded systems that are to be programmed in Ada. The essential requirement of such a simulator is to represent the multitasking aspects of the program, as opposed to the other computational functions. There are then two basic design decisions for such a simulator. One is whether to use the analytic or the discrete event approach in the construction of the simulator. The second is whether to provide the designer with a generalized language that would enable him/her to express and to structure the parallel processes in a manner that would be independent of the way it might be most natural for them to be coded in Ada or to create a simulation language that conformed to all of the normal Ada constructs. The decisions made in the development of the ESDS for each of these cases is discussed below.

6.1 ANALYTIC VS. DISCRETE EVENT SIMULATOR

Analytic simulation is based upon the use of classical queueing theory. The system is viewed as an assembly of servers, each of which has a service time that is represented as a random variable of some analytic distribution function. The essential parameters that enable a solution to the expected queue wait time for any of these servers is the mean and variance of these functions. One problem with this approach is that assumptions are made about the fact that the service times are able to be represented by such a function. Another is that the queue length formulas make assumptions about the pattern of incoming calls for service. One relates to independence of arrivals to the queue, i.e., that the arrival of one caller on the queue does not have a functional dependency on the previous caller. An example of such a dependency is a periodic call (i.e., every n seconds). Also, a singular call (one time only) or arrivals that do not have a random pattern (such as three calls in quick succession, none for a long time, and then one more), are more difficult to represent accurately, in terms of the assumptions made about arrival patterns to the queue.



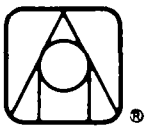
The discrete event method makes no assumptions whatever about either the nature of the server or about the calling patterns over time to the server.

The principal advantage of the analytic approach is that it runs much faster. The disadvantage is the flexibility of representation of the server and the call arrivals. Another advantage of the discrete event simulator, which is being exploited in the ESDS, is that the user can interact with the simulation. This can be done because there is a simulation clock that can be interrupted at any time, under program control. When this is done, the entire state of the system -- sizes and members of queues, values of variables, and the entire history of the system up to this "time", are accessible; hence the user can interact in a "real simulation time" with the simulation as it is running. Also, to facilitate this process, the simulation clock can be made to run at various speeds, under user control, relative to the wall clock, so that the user can adjust the flow of events in the simulation to any real reaction time desired. The simulation can also be run in a stop frame mode, one event at a time, or one time unit at a time. In principle, it can be stopped, moved back in time and then run forward again, like a movie. The current version of the ESDS does not allow this, however, because far more historical status data must be stored than is required for the normal simulation report.

The decision was therefore made to implement a discrete event simulator. The slower running speed is partly compensated by today's high speed processors.

6.2 GENERAL VS. ADA-DIRECTED MULTITASKING

The decision to make the ESDS Ada-directed relates to the discussion of Sections 1.2 and 2.2, wherein it is proposed that the methodology be infused with the structure and the design principles of the Ada language. In addition, it will form a much better communication link between the designer and the implementer. The ESDS therefore maps all of the Ada multitasking constructs onto the simulator and enables them to be easily expressed by the designer



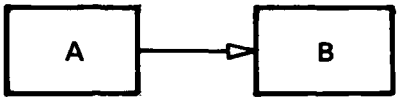
through the model building language of the ESDS, as will be described further on. Figure 1-5 presents a succinct illustration of the Ada multitasking capabilities. If the designer wishes to employ multitasking capabilities that are beyond those of Ada, such as the one discussed in Section 2.4.2, but which can nonetheless be implemented by the programmer either through the VRM concept introduced in Section 2.4.2 or via his/her own custom design, then these capabilities would have to be added at a later time to the ESDS.

6.3 ESDS REPRESENTATION OF A MODEL

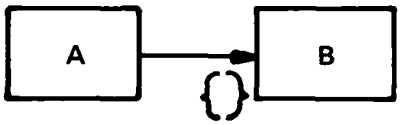
The ESDS model represents the inter and intra task view of an Ada program. It could be used to represent the intersubprogram view, but this is not considered essential to its purpose, which is to enable the designer to determine whether a given multitasking design will provide a desired response time. This is based upon an allocation of functions to tasks, an interconnection of tasks, and some internal task design, which will be discussed. The intrasubprogram view is definitely excluded, because this would constitute the detailed design of the program itself. Note that in Figure 3-1 of Section 3.2, the use of Modeling (and its consequent database layer) is positioned between the high and low level design. Thus, high level design, with respect to multitasking, is here defined as the inter and intra task views, possibly the intersubprogram view, but not the intrasubprogram view.

6.3.1 The Intertask View

The intertask view can be represented by the conventional block diagram plus a few additional conventions to enable Ada's various modes of intertask communication and synchronization to be represented. The first distinction to be made is between the initiation of a task vs. the synchronization between tasks. Figure 6-1 illustrates a notation that might be used to make this distinction. In 6-1A, task A initiates task B; and in 6-1B, task A calls and synchronizes with task B. As was discussed, there are a number of synchronization modes. These would be distinguished by a symbol chosen from within the braces, to be discussed later.



A INITIATES B
(1A)



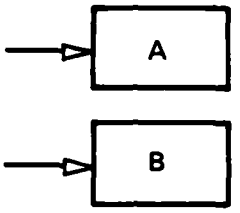
A CALLS AND SYNCHRONIZES WITH B
(1B)

Figure 6-1. Task Initiation and Synchronization

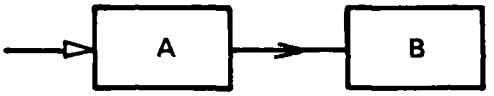
The next issue to be resolved is whether a task is monolithic in the sense that a task equals a block (in the diagram), which maps onto a single system function. A task with one or no entries is monolithic in this sense. In the simplest case of no entries, no other task interacts with it, though data may still be communicated via global variables to an outer block.

Figure 6-2 illustrates the no-entry task. In 6-2A, task A is completely independent of task B. In 6-2B, task A passes data to task B, where B is global to A. If A is the initiator of B, then there would also be an open arrowhead as in 6-1A. If not, then it would appear exactly as in 6-2B.

Figure 6-3 illustrates the one-entry task. The solid arrowhead indicates that A calls B1, which is the (single) entry of B. The "S" means that the executions of A and B (B1) are synchronized -- i.e., they will rendezvous. The directed arrows in the middle of the line indicate data flow between A and B as passed parameters. From the viewpoint of A, B performs one function, though code may be executed in B that precedes or follows the scope of B1.



NO ENTRIES, NO DATA COMMUNICATION
(2A)



NO ENTRIES, A COMMUNICATES DATA TO B
(2B)

Figure 6-2. No-Entry Tasks

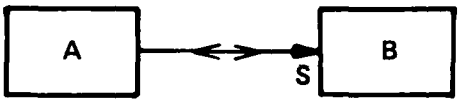


Figure 6-3. Task B Has One Entry

The problem arises with Figure 6-4, which illustrates a multiple-entry task. Task A can select any of a series of entries (i.e., procedures) within task B. The basis for this selection may be fixed within A -- i.e, A or some particular call within A, always calls Bk. Alternatively, it could be conditional -- e.g., if velocity is greater than Mach 1.4, then call Bj; else call Bk. This is different from Figure 6-5, where A can call n different tasks via n separate calls. In principle, this implies n independent and parallel tasks. In Figure 6-4, the entries can only run alternately. In practice, the n tasks of Figure 6-5 would not run concurrently if all of the calls were synchronized (which is the only kind of call in Ada). However, if we look at the more general scheme, a call could be non-synchronized, or concurrent, as indicated by the "C". In a synchronized call, A does not continue to run until the Bk call is complete. In a concurrent call, A continues to run while Bk is running or while its call is queued before Bk. Concurrent task calls will be discussed later. They are not implemented in Ada.

One further complication is added, as illustrated by Figure 6-6. If A calls C1 and B calls C1 before C1 finishes servicing the A call, then the B call is queued. All subsequent calls are queued and then serviced FIFO in Ada. However, in principle, other queue service disciplines are possible, such as by priority of calling task or by the value of a variable passed as a parameter. Another option includes a preempt, which immediately places the call at the head of the queue. Were queue service disciplines other than FIFO to be implemented, this too would have to be indicated in the diagram.

In summary, a block diagram can be constructed that presents parallel processes; but to map into an Ada program architecture, the five constructs of Figures 6-1 to 6-6 must be recognized.

Added to this are two other related concepts. One is what might be called the cloning effect. A task may be singly or multiply initiated. This is different from multiple calls. A multiply initiated task means that n versions

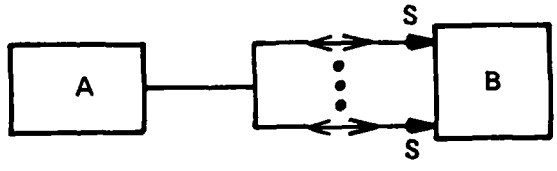


Figure 6-4. Task B Has Multiple Entries

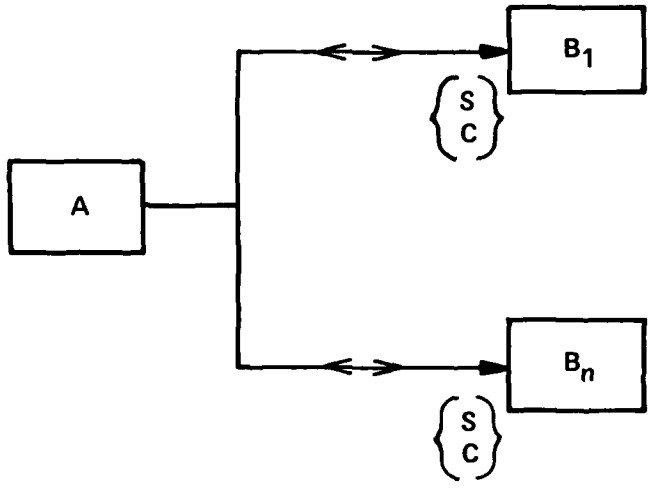


Figure 6-5. Task A Calls n Separate Tasks

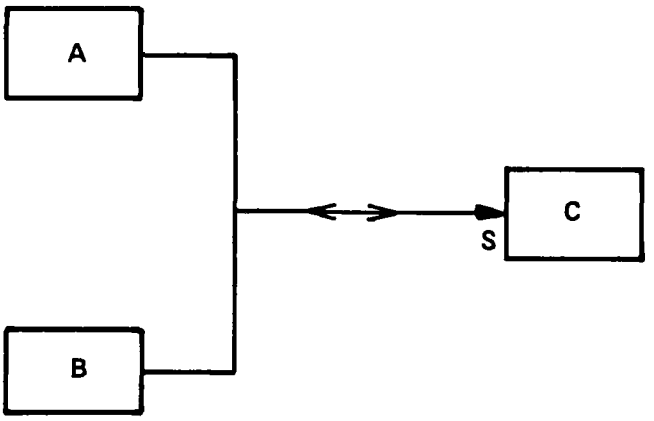


Figure 6-6. Multiple Calls to C

of the same task are running in parallel. For example, assume that a task is a signal processor and that, at a given time, there are n different signals to be processed. In principle, the designer could initiate n of these tasks (all identical) and assign one to each signal. He could then abort a task at the conclusion unless there was another signal to be processed. Alternatively, all the signals could be processed, in turn, by a single task. If there is only one physical processor, then the practical effect of the two approaches is the same because the n tasks are going to share the same processor and be executed in series anyway. There is a real difference in a multiprocessor system, because the first case represents n true parallel processes and the second represents a single process that is used sequentially. It is questionable whether task cloning should be considered at the block diagram level of design. It is probably better to leave this decision for a later, more detailed step.

The second additional concept involves the two ways in which data signals are processed. One is by interrupt; the other is by being polled. In the former, the device that presents the signal on an input line or channel raises an interrupt signal in the processor that causes the program to transfer immediately to a particular location to begin processing the signal. In the latter, the processor polls the line for data; if present, it processes the data.

In some cases, the polling is performed at regular intervals under clock control; in other cases, the polling is triggered by some event or when control happens to return to some level of executive control.

If the designer must cope with these concepts as well in the block diagram, then a means of expressing them must be found. Figure 6-7 presents a set of graphic symbols that would be required to express all of the above-described concepts, except task cloning.

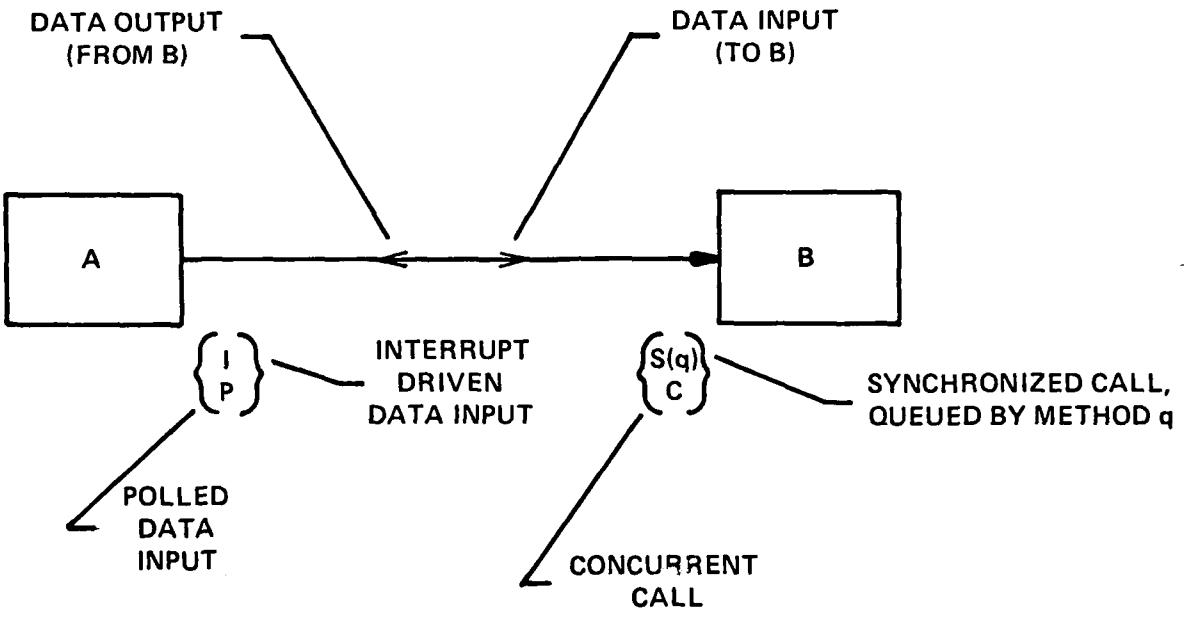


Figure 6-7. Summary of Synchronization Symbols for Block Diagram Graphics

6.3.2 The Intratask View

The ESDS models each task by defining a series of events that comprise it. There is an initiating event, and then each event, from the initiating event onward, triggers another event. In certain cases, as will be seen, an event can trigger several other events or possible events. Each event in the

model is characterized by two numbers. One is an event type and the other is a unique serial number within the total model. Table 6-1 identifies the nine event types in the ESDS.

Table 6-1. Event Types

1 = Task initiation
2 = Sequential processing (entry)
3 = Sequential processing (non-entry)
4 = Call (Task entry)
5 = End of entry
6 = End of task
7 = Rendezvous (accept or select)
8 = Delay (non-select)
9 = Delay (select)

There are four cases in which an event can be succeeded by two or more different events. One is if an event concludes by initiating a task. In this case one successor event is the initiated task, the other is the normal successor within its own task. In this case both successor events become active (i.e., they actually occur). Event types 1 to 3 can cause such a task initiation. The others cannot, because they do not represent processor activity.

The second case, event 4, is the call. One of its successors is the entry (a type 2 event) of the called task; the other successor is the next event in succession within the task of the call after the call has been serviced. In this case the two successor events are not concurrent, but rather occur in sequence in accordance with Ada's synchronization rule.

The third case is a program branch, as would be created by an *if* or *case* statement. In this case, the model builder must assign a probability to each branch, where the sum of the probabilities for all branches is one. ESDS will select one of the branches, based on the probability distribution. Again, only event types 1 to 3 can branch in this way.

The fourth case, event 7, is at a rendezvous. The Ada *select* statement permits a variable number of successors of event types 2, 6, or 9.

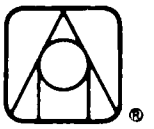


Figure 6-8 presents the block diagram for a simple three-task example. Tasks 1 (T1) and 3 (T3) are elaborated and thereby initiated by the normal program execution. Task (T2) is dynamically initiated by T1, as indicated by the dashed arrow. There is an input to T1 and an output from T2. This represents the entire intertask view.

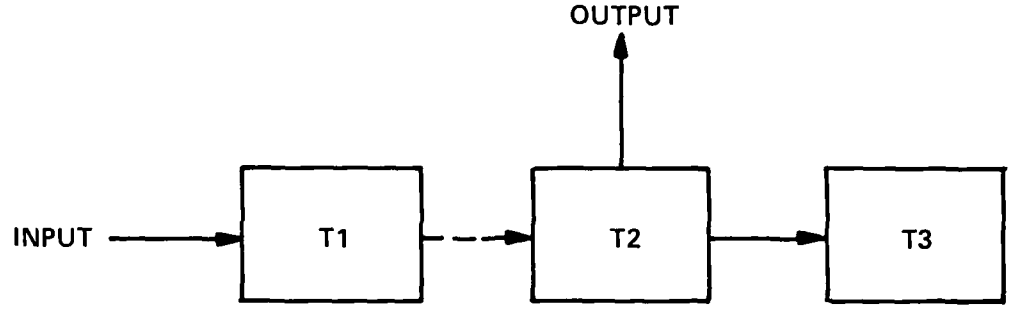


Figure 6-8. Block Diagram for a Multitasking Model

Figure 6-9 presents the corresponding intratask view, in the form of what is here called an Event Diagram. The event type numbers are shown in parentheses in the event diagram, and the serial event numbers are entered above and to the left of the parentheses. All of the quantitative information pertaining to the model, including the event serial number, type, execution time, and successor event(s), is contained in the Event File, illustrated in Table 6-2. Also shown in the figure is a priority for each task; the higher the number, the higher the priority. Event 1, in the example, is the initiation of a task simply by program elaboration. That is, when the program starts, the task also starts to run. The same is true for event 10. The simulator assigns serial numbers, dynamically, to each running task. Thus, events 1 and 10 have initiated tasks 1 and 3. In actuality, the task is initiated when control reaches the task body elaboration in the Ada program. Each task is assigned a *status* by the simulator. At this point, tasks 1 and 3 have a status of *run*, but in fact one or both

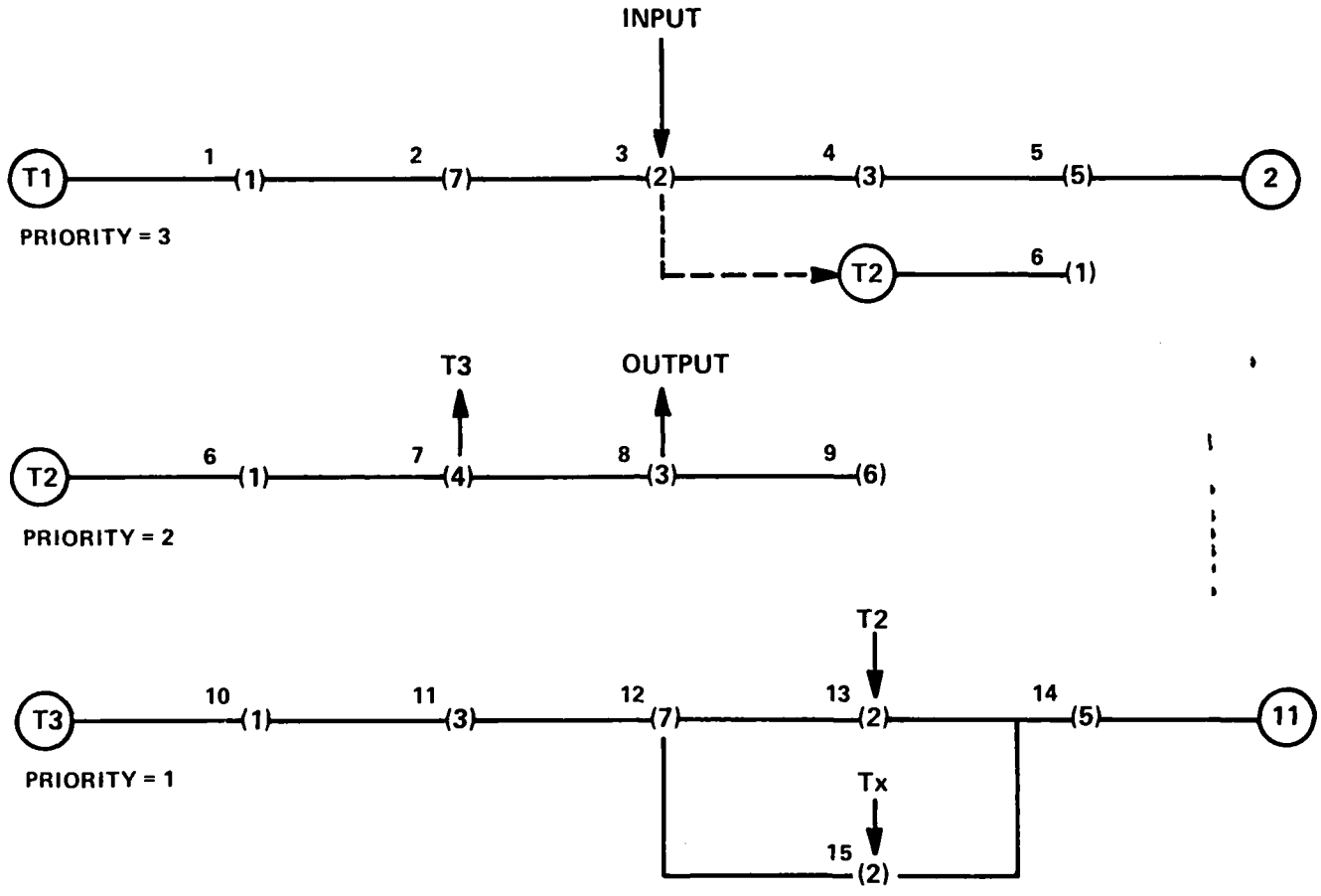


Figure 6-9. Event Diagram for a Multitasking Model

will actually be *executing* if there is a processor (CPU) available. The ESDS distinguishes between the status of *run* and *execute*. In a single processor system, only one task can have a status of *execute*, while many can be in a *run* state. In a multi or distributed processor system, multiple tasks can be *executing*, though some may still be in a run state if there are insufficient processors or if a processor in a distributed system is itself multitasking.

Table 6-2. Event File

Event No.	Event Type	Priority	Execution Time	Next-event-1	Next-Event-2
1	1	3	3	2	
2	7	3	-	3	
3	2	3	5	4	5
4	3	3	4	15	
5	1	2	4	7	
6	1	1	2	8	
7	4	2	-	13	10
8	3	1	2	9	
9	7	1	-	10,11	
10	2	1	7	12	
11	2	1	6	12	
12	5	1	-	8	
13	3	2	7 (Output)	14	
14	6	2	-		
15	3	3	0	2	

A task may be in one of the five states shown in Table 6-3.

Table 6-3. ESDS Task States

1. Suspended in delay
2. Suspended at rendezvous
3. Suspended on a call
4. Running
5. Executing

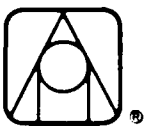
Referring to Figure 6-9 and Table 6-2, we see that the successor event to 1 is 2, which is a rendezvous resulting from either an Ada *select* or *accept* statement. In Ada, a rendezvous is made with a task by means of a task *entry*, which can be called by another task or by an external interrupt. Task 1 is shown to have one entry at 3. Event 3 is the entry for an INPUT. Ada calls this type of entry an *interrupt*. Each entry is preceded by a rendezvous event (type 2). From Table 6-2, event 1 runs for three time units and arrives at the rendezvous (event 2). Event 2 has no predetermined time. It waits until an input arrives. At that time, event 3 occurs for five time units. This is the sequential processing of the entry code. It may have any internal structure involving subprogram calls, branches, and loops.

Next we see a branch in the control flow at the end of event 3. One is the beginning of event 4; the other is the beginning of event 6, a dynamic task initiation. At this time, there are three concurrent tasks. Event 6 (Task 2) runs for four time units and reaches event 7, which is a call to an entry in Task 3. This entry is event 13, which has its rendezvous (select or accept) at Event 12. Task 2 is suspended after the call from 7 to 13 until the processing of entry 13 is complete, whereupon Task 2 continues at event 8. The output results from event 8 processing after seven time units.

A mutiple entry is illustrated in the figure and in Table 6-2 as event 12, which would represent a select statement, with alternate entries 13 and 15. Entry 13 is used but entry 15 is not, so as not to further complicate the example. In an actual program, entry 15 would also be used, or it would have no meaning. A call from some task Tx is simply shown here.

6.4 THE USE AND RATIONALE OF THE ESDS

A number of design decisions were implied in the previous example. First was the decision to dynamically create new instances of Task 2 (which would receive unique numbers 4,5,6... if old ones had not reached termination (event 9)). This would make sense only if there were multiple processors. In a single processor system, a better design would be to elaborate Task 2 at the

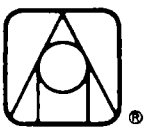


beginning and to let it run. Then a call from Task 1 to an entry in Task 2 would initiate the process of event 6, and the creation of multiple tasks would be substituted with a queue on a single task. The ESDS would enable both of these designs to be modeled and compared.

There is also a presumption that Task 3 does something independently of Task 2 at event 11, such as sampling sensors for data and updating a data structure (even though the sensor input has not been shown, for simplicity). Otherwise, Task 3 could have simply been a subprogram of Task 2. Again, this structural change is rapidly made with the ESDS to make a comparison.

For each program design, the system will be stimulated with a series of INPUTs and, depending upon the arrival distribution and intervals, the response of the system (to each input) will vary. The ESDS produces reports not only on the response times, but on the queues that develop at each task entry. It also generates a series of performance measures on the tasks and events. These are discussed in more detail later in the paper. There is an input generator that will generate different inputs according to periodic or random patterns, with varying frequencies and mean interval times. The user can also superimpose (or use exclusively) manually generated inputs.

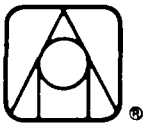
Normally, the exact execution time would not be known by the designer, because the actual program has not yet been written. Therefore, an estimate is made. The model is used to determine whether an adequate response is achieved under all tested input conditions. If the design is satisfactory, then the estimated event times can become specifications to the programmer. The simulator can also be used in this way to determine maximum allowable execution times for subprograms, in order that a given model achieve required response times. As the code is written and tested (in the target machine), the real execution times can be substituted in the model, and the simulation can be run again. Thus, the model is used initially to design the task level program architecture and to specify required execution times at the intratask level. Then it is used



to monitor and verify the implementation and to alert the designer if the actual intratask execution times vary to the point of rendering the overall system response inadequate.

The ESDS provides the designer with a high degree of precision in the operational analysis of the simulated program. First, he/she can determine which input-output pairs do not respond as required. Second, he/she can determine which task, task entry, or event is causing the bottleneck. The reasons may be low task priority or excessive execution time of critical events. A variety of solutions may then be considered. The first is to change relative task priorities. The second is to restructure tasks by separating noncritical (i.e. low priority) from critical (high priority) subprograms. The third is to reduce execution time of critical events. The fourth is to consider whether an unconditional task call can be changed to a conditional call. This would have the effect of not performing a certain task at a given time if the wait time were too long, which in turn has the effect of reducing queuing on that task. Alternatively, there may be hardware remedies to consider, such as additional processors in a multi or distributed processor configuration or a faster processor.

The ESDS also provides efficiency measures for each task or event. These are numbers between 0 and 1, where 1 is maximum efficiency and 0 would mean "locked out". These numbers can be used as a general measure of design efficiency so that even if all response times were acceptable, one might still want to further optimize the system's performance by improving the efficiencies of selected tasks or events. The purpose of this would be to provide a greater safety margin in the event that future requirements change. These are discussed further in the next section.



6.5 STRUCTURE OF THE ESDS*

Figure 6-10 presents an overview of the system. It consists of three programs: BUILD_MODEL, GEN_INPUT, and SIMULATOR. BUILD_MODEL builds the model by generating two files. The EVENT file contains the events and their sequence relationships. The INPUT_ASSIGNMENT file contains a description of each input to the program, in terms of the event (or events) that it triggers and the output(s) that result. BUILD_MODEL also produces a report that documents the model.

GEN_INPUT creates an INPUT file containing a series of inputs and the time at which each is to occur during the run of the simulation. Inputs can be generated in two modes: *manual* and *automatic*. The *manual* mode represents a single input that the user enters with its time of occurrence. The *automatic* mode enables the user to specify parameters to a generator (procedure) that will automatically create inputs according to a selected generator algorithm. Two algorithms currently exist; others can be added to satisfy special requirements. One is to create an input of a particular type *periodically*, where the type and *period* are entered as parameters. The second is to generate n inputs randomly, where a *type*, *period*, and n are given and where the n inputs are generated at random times within each interval.

Finally, the SIMULATOR program runs the simulation. It uses three files internally: the Task Status Table tracks the status of each task; the Queue File maintains all of the necessary task queues; and the Statistics File maintains the simulation run time statistics for the report.

The SIMULATOR produces a three-part statistical report. The first part presents the response time for each associated input-output pair. The second part contains a time domain histogram of the length of each queue in the Queue

* See Reference 23 for a more detailed description of the ESDS.

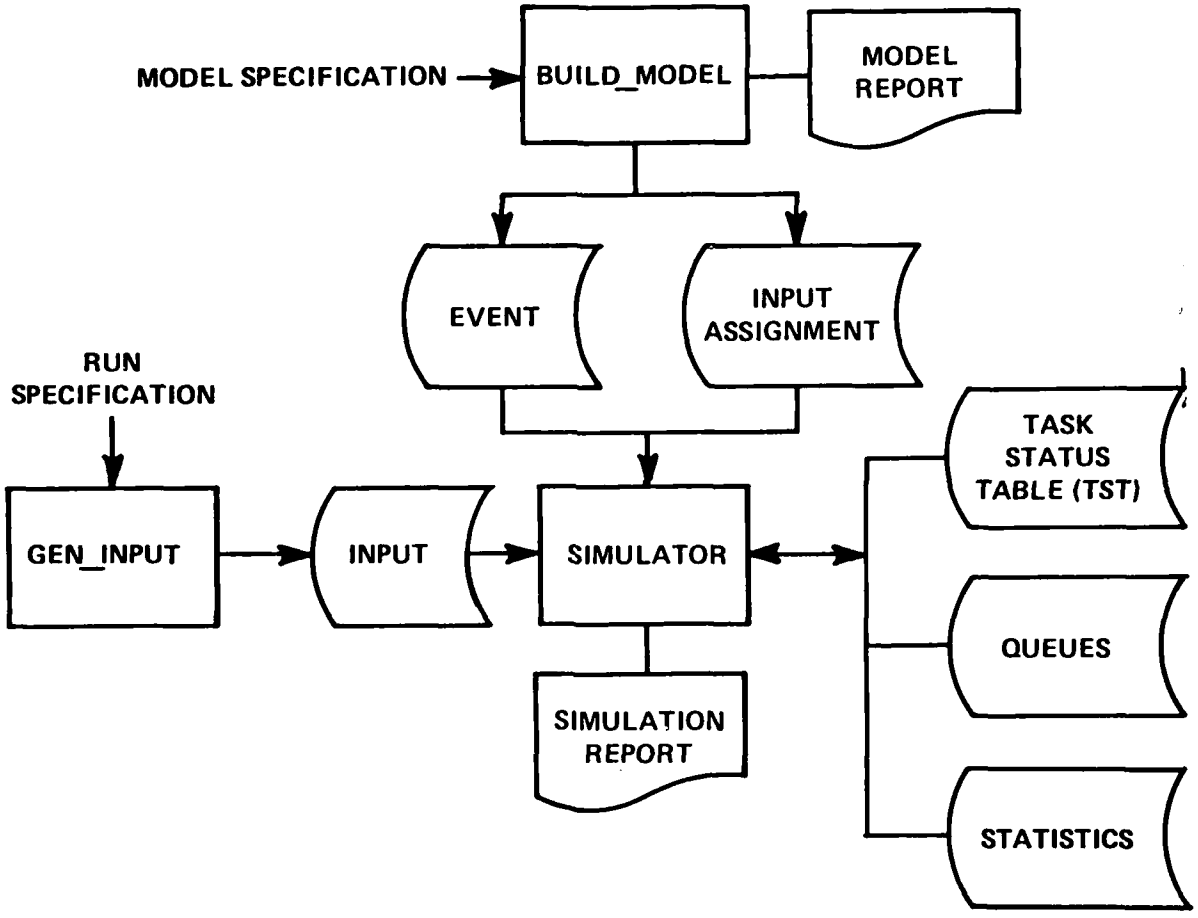


Figure 6-10. The Embedded Software Design SIMULATOR (ESDS) System

File and the average queue length and wait time. The priority of the task serving each queue is also given in the time domain, because the priority in Ada can change based on that of the calling task.

The third part presents a series of operational efficiency measures, which are defined in terms of the state variables of Table 6-3 and upon the following response parameters:

1. Required response time
2. Minimum possible response time
3. Actual response time

The operational efficiency measures are then defined as:

(1) *Event efficiency* is the ratio of the event execution time to event execution time plus run time.

(2) *Task Efficiency-1* is the ratio of the task execution time to the task execution time plus run time.

(3) *Task Efficiency-2* is the ratio of the task execution time plus run time to the total active time of the task (which is the sum of all five states).

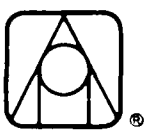
(4) *Response Efficiency* is the ratio of the minimum response time to actual response time. The system also reports any actual response time that exceeds the required response time. These are called *Response Failures*.

(5) The *Response Efficiency* is based upon the system characteristics as well as the specific set of inputs for a given run. It is possible that there may be Response Failures with high Response Efficiency, and, conversely, there may be no Failures with low Response Efficiency, because a Failure is simply defined as the quantity (Required Response Time - Actual Response Time) being negative. Another measure that is more useful than Response Efficiency is *Response Elasticity*. This is a number that has meaning when average (Actual Response Time) is less than average (Required Response Time):

Let A = average (Actual Response Time)

R = average (Required Response Time)

E = *Response Efficiency*



Then,

Response Elasticity = (R - A)E/R.

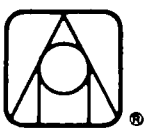
This number varies between 0 and 1, but would never reach 1, because A cannot be zero. A high elasticity for a given input sequence means that the system is far from being saturated and could respond adequately even if the inputs were to arrive more frequently.

This report, along with the MODEL REPORT from BUILD_MODEL, enables the designer to determine the cause of inadequate input-output response time in terms of (1) long queues, (2) low event, task or response efficiency, and (3) excessive event execution times. Long queues are remedied by increasing the task priority or, if the task contains multiple entries, by separating one or more of the entries, if possible.

6.5.1 Interactive Operation of the ESDS

A discrete event simulator has a software clock that tracks simulation time. Appendix B presents a diagram that defines the basic executive control of the system, along with the complete PDL that details the program logic. The executive control is a loop that, with each iteration, decides what event is to occur next. Events, as used here, refer both to the task events described above as well as to all input stimuli to the system. Each of these events has an associated event time, and the executive has simply to select that event with the next higher event time as compared with the current simulation clock time. It then advances the simulation clock to the value of the selected event time. Here the simulation clock does not "tick" like an ordinary clock in equal time increments. With each executive iteration, there is thus an opportunity to stop the processing and allow the user to interact with the system in its current state. This feature has been used to develop a method of interactive design.

Two interactive modes are to be provided. In one, the simulator runs continuously, but at a slower speed, controlled by the user at the terminal, so that the changing states of the system can be viewed dynamically. This is done simply by executing a delay at the time the executive gets control with each loop iteration.



The second is a "stop frame" mode. Whenever the simulation clock changes, the simulator stops and displays the current state of the system. The user can also change between these two modes during the run. The system state is viewed by five summary and detail tableaus selected by the user. When viewed in the continuous mode, part of the tableau remains constant and part changes. Appendix C provides more detail on the content and format of these tableaus. They are briefly described below:

(1) *Task Summary*: Summary information on every task, or a user selected set of tasks, is presented. This information includes the task status, priority, efficiency measure, number, and size of queues, and average I/O efficiencies, elasticities, and failures.

(2) *Task-I/O Detail*: For a particular task, the I/O efficiencies, elasticities, and failures for every I/O pair are presented.

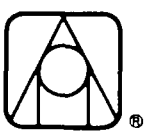
(3) *Task-Event Detail*: For a particular task, the event efficiencies and queues are presented for every event in the task.

(4) *Queue Summary*: Summary information on every queue is presented. This includes queue type, size, average wait time, and the current efficiencies of the events and tasks involved. The user can select the queues to be viewed.

(5) *Queue Histogram*: A time domain histogram is presented in a tabular format for each queue. As in tableau (4), the user can select the queues.

It is expected that the interactive design modes will enable the designer to identify problems more rapidly and precisely than in the end of run report mode. The latter will still be available and is more useful in cases where the designer wants to work with the simulation results away from the computer.

At the present time, only tableau (1) has been implemented. The other four would represent future work.



In conclusion, use of the ESDS assumes that the designer can represent the program in terms of events, estimated event execution times, and event sequencing and interconnectivity relationships. The simulator then enables the designer to determine, fairly precisely, response characteristics of the system. Analysis of the quantitative information in the SIMULATOR Report provides him/her with specific alternatives for design improvement.

6.6 EXAMPLE OF AN ESDS RUN

Using the example of Figure 6-9, an input file was prepared and run. Since there is only a single input and output type in the example, the only relevant input information is the arrival time of each input. These occured at time units: 2, 25, 55, 90, and 120. The absolute minimum response time of the system is 23 time units, but this is predicated on the availability of multiple processors and truly concurrent tasks. If too few processors are available, then the actual response time may be greater than 23 time units, depending on the arrival rate of the inputs. The mean arrival time interval for the inputs in the test run was 29.5, and the simulation was run for a single processor system. As will be seen, there was still considerable queuing, because of the fact that there are multiple tasks and only one processor.

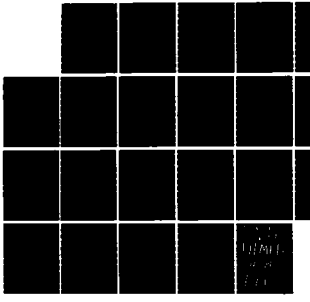
Table 6-4 presents the results in terms of the I/O response times and efficiencies.

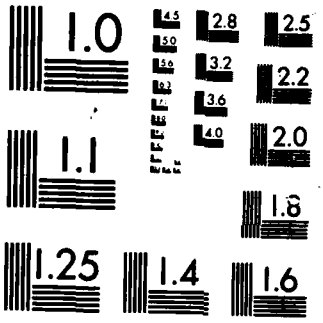
Table 6-4. I/O Response

	TIME			
	INPUT	OUTPUT	ACTUAL RESPONSE	MINIMUM RESPONSE	I/O EFFICIENCY
	2	71	69	23	0.33
	25	105	80	23	.28
	55	139	84	23	.27
	90	159	69	23	.33
	120	175	55	23	.41
Average			71.4	23	.32

AD-A165 497 DESIGN METHODOLOGY FOR EMBEDDED SYSTEMS(U) ANALYTICS 2/2
INC WILLOW GROVE PA D LEFKOVITZ 15 NOV 85 TR-1500.21
N62269-80-D-0025

UNCLASSIFIED F/G 9/2 NL





If the Required Response Time of the sytem were 75, then an I/O Failure alarm would have occurred for INPUTs 25 and 55. The Response Elasticity is only 1.5%, because the I/O Efficiency is only .32 and the Required Response Time of 75 is only slightly larger than the Actual Response Time of 71.4; however, if the Required Response Time were 100, then the Response Elasticity would be 9.2%. Even this is a rather low margin of safety. Other efficiency factors, to be examined later, will also be seen to be low, indicating that this system is barely adequate to handle the presented input load.

Table 6-5 presents the Task and Event Efficiencies.

Table 6-5. Task/Event Efficiencies

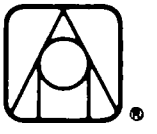
TASK	EVENT	EVENT TYPE	TASK PRIORITY	EFFICIENCY	
1			3	1	
	1	Initiation		1	
	3	Interrupt		1	
	4	Sequential		1	
2.1-2.5			2	.24	(Avg)
	6	Initiation		.25	
	8	Sequential		.43	
3			1	.28	
	10	Initiation		.09	
	11	Sequential		.11	
	13	Entry		.71	

Several points of interest can be noted here. The Efficiency of Task 2 is given as an average because five such tasks were generated in the course of the run, but, despite its higher priority, it had a lower efficiency than Task 3. This is because there were multiple instances that shared the same priority level, and all contended for the single processor. Also, even though Task 3 has a lower priority, it inherits Task 2's priority in the entry of event 13. This is also evidenced by the significant difference in the Efficiencies of events 10 and 11 vs. 13. It can also be noted that the highest priority task, 1, has an Efficiency of 1, as do all of its events. This explains, as will be seen, why no queue developed at event 3. If either the priority of Task 1 were lower or

inputs had arrived faster, a queue would have developed there as well as at event 13.

No queues developed at entry (interrupt) 3, because it was such a high priority that whenever an input appeared, it was processed immediately through event 3 for five time units, and then through 4 for nine time units, which was given precedence over the parallel task 3. Thus, it took fourteen time units for task 1 to recycle back to its rendezvous at event 2, and since the shortest time between inputs was 23 units, a queue never developed. However, a queue does develop at entry 13. Figure 6-11 presents a histogram of that queue. Multitasking system throughput analysis is difficult by classical queuing theory techniques, because of the interactions among the priority algorithm, and the effect of various Ada commands involving rendezvous, suspension, delay, and "when" guards. Discrete event simulation, however, is able to take all of these factors accurately into account.

Thus, the histogram in Figure 6-11 shows the queue size at entry 18 in its precise time relation to each input and output. We see that even the first input at time 2 is queued at time 21, even though the next input doesn't occur until time 25. The reason is the relative priority of the tasks. Referring to Fig. 16 and Table 8, we see that the first input is accepted at time 3, after event 1 has run for 3 time units. Task 1 has the highest priority, so event 3 executes for 5 time units, whereupon task 2 is created; however. task 1 continues to execute for another 9 units at event 4. Thus far 17 time units have elapsed. After event 4, task 1 returns to its rendezvous at event 2 and gives up control to task 2, which has the next highest priority. Event 6 runs for 4 units and then calls entry 13 of task 2 at event 7, but task 2 is not yet at its rendezvous. It has the lowest priority and must still process events 10 and 11; therefore, the call is queued at 21 time units. Tasks 1 and 2 are now both suspended, so task 3 executes events 10 and 11 for 4 time units, whereupon it arrives at the rendezvous (event 12), and the call is immediately accepted by event 13 at 25 time units. The histogram of Figure 6-11 shows the queue with one entry at time 21 and with the service of the call at time 25 reducing to no entries. Further on, as more inputs appear and as additional task 2s are



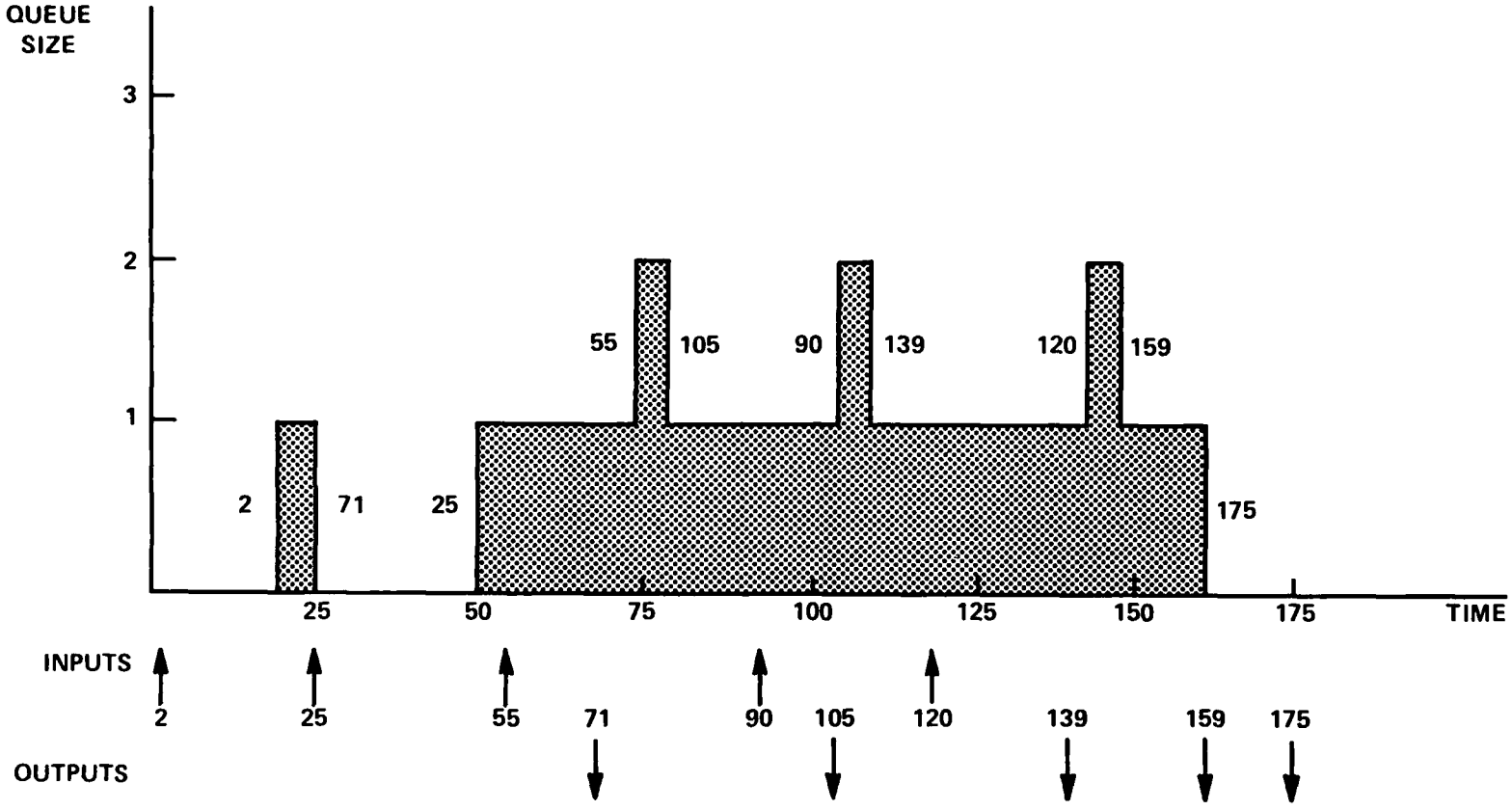


Figure 6-11. Queue Histogram for Entry 13

created, the system is not fast enough to clear the queue completely until after the last input at time 120. With a minimum service time of 23 units (achievable only with multiple processors) and an average input arrival interval of 29.5 units, it would be difficult to predict that the average response time would be 71.4 time units without this type of simulation.

6.7 FUTURE WORK

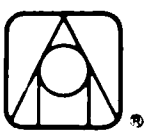
6.7.1 Interactive Design

As discussed above, the remaining four tableaus must be implemented for the interactive design mode.

6.7.2 Reverse Modeling

Section 3.2.3 introduced the concept of the "reverse" model. The events of Table 6-1 and the intratask structure illustrated in Figure 6-9 show how the model maps directly onto Ada's multitasking language structure. In principle, one should then be able to map the Ada source code that generates this structure (i.e., the task specification, the select and accept structures, and the calls) back into the model specification. This we would then call the reverse model. It is analogous to disassembly, wherein one translates from a lower to a higher level language.

As discussed in Section 3.2.3, reverse modeling has several purposes. First, it can be used to optimize the design of an existing program, for which the ESDS had not been used originally. Second, it can be used to verify the coding of a previously modeled design. All of the quantitative measures (I/O response times, efficiencies, queues, etc.) of the forward and reverse models can be compared. They cannot be expected to compare exactly, but gross discrepancies will indicate possible areas of erroneous implementation. The most likely problem will be in the actual execution time of certain events. These problems would be resolved by (1) accepting the implemented value if the



response time requirements are still met, (2) improving the implementation if they are not, (3) determining whether a program architectural redesign is needed, or (4) determining that the specifications cannot be satisfied without hardware improvement.

6.7.3 Analysis of Operating Systems

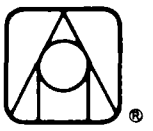
The current ESDS version isolates the code that deals with the hardware configuration and multitasking functions that may be performed by the operating system. At present only a primitive multiprocessing algorithm has been implemented. A future improvement will be to write procedures for multiprocessing and distributed processing systems. These can be written to represent various hardware configurations and operating system algorithms. In this way the ESDS could be used not only as a design tool for Ada programs in a given hardware/operating system environment, but could also be used to test proposed designs of hardware configurations and operating systems (with respect to their multitasking strategems).

6.7.4 External Devices

The inputs to the ESDS are simply represented as an occurrence of a signal at a given time. There is no ability to characterize the behavior of devices such as disks, tapes, telecommunications, etc., so as to extend the simulation capability to the operation of these external devices. Thus, the repertoire of event types must be expanded, or a general purpose simulator interfaced to the ESDS. In the former case, the external devices would be modeled by the use of new event types and appropriate parameters associated with them. In the latter case the devices would be modeled in the language of the interfaced general purpose simulator.

6.7.5 Testing

The greatest need at this time is to validate the principles of the ESDS by applying it to some realistic problems, and then to thoroughly evaluate it.



7. CONCLUSIONS

The major conclusions of this research are:

1. Communication among persons working in the same lifecycle stage and between the disciplines of each successive stage is one of the most important problems to be solved. This is best done by documentation methods that are at once structured and flexible. The structure enhances completeness and consistency. The flexibility enables natural language to be used, when required.

   At the requirements specification stage, SREM [3] or the NRL method [14] are quite suitable. At the high level design stage there are a number of techniques available, including Structured Design [8] and SADT [5]. At the low level design stage, the best approach appears to be the PDL [9].

2. The design principles and the structure of Ada should be integrated into the entire process, as a means of further enhancing communication. Toward this end, an Ada oriented view of a system has been proposed that leads to a series of Program Architecural Diagrams (PADs). These represent Ada's inter and intra views of tasks, packages, and subprograms. Also, the inter-relations among five basic Ada components can be viewed as a network. These components are types/subtypes, data objects and the three program units, tasks, packages, and subprograms. Relations among them that define the branches of the network are: where defined, declared, used and called, and their inverses.

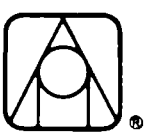
3. The concept of iterative design was endorsed. This recognizes the fact that a lifecycle stage usually is not complete and consistent, in practice. Therefore, feedback mechanisms should be built into the methodology to facilitate recycling between lifecycle stages.

4. Another tool was introduced as part of the design process, falling between the high and low level design. This is a simulator, called the Embedded Software Design Simulator (ESDS) [20]. Its purpose is to enable the designer to test several Ada multitasking program architectures to determine whether critical response times can be met. It is based upon the high level design plus certain execution time estimates. This enables a good high level design to be established before committing the greater effort involved in low level design. The execution time

estimates would also become programming specifications. The ESDS would also become an important element in the iterative design process. The model can aid in the validation of the implementation through a process called reverse modeling, and it can be used to perform a large number of more accurate tests after the implementation has determined the actual execution times of certain program elements.

5. All of the tools discussed above should be integrated into a single Database Management System constructed for this purpose. It would enable both project management and technicians to readily access information about any of the stages, and, when authorized, to update it. Also, project management could track progress through both the standard milestone mechanisms as well as inventory, monitor, and examine the actual development documents.

6. Even though Ada may be adopted as the standard compiler language for embedded system implementation, multitasking capabilities that have been excluded from Ada should continue to be investigated. These include run time control over task priorities, queue management disciplines other than FIFO, and concurrency for calling tasks. This could be accomplished through a combination of special operating system calls and the writing of a package of procedures to perform these functions, similar to the way MASCOT [11] enhanced a special version of PASCAL and GASP [21] enhanced FORTRAN. This project has generically called such a capability a Virtual Realtime Machine (VRM), because it can create an apparent machine with greater multitasking capabilities than are inherent to the basic set of Ada commands.



REFERENCES

1. Lefkovitz, D. "Embedded Software Development: Tools vs. Methodology," NADC Contract No. N-62269-D-0025, Task Order 21, May 15, 1985.

2. Lefkovitz, D. "Design Methodology for Embedded Systems: Focus on Parallel Processing," NADC Contract No. N-62269-D-0025, Task Order 21, December 31, 1983.

3. Alford, M. W., et al, "Software Engineering Methodology. Report No. CDRL C005," TRW Defense and Space Systems Group, August 1, 1977.

4. Teichroew, D. and Hershey, E. A., "PSL/PSA: A Completer Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Trans. Software Eng., January 1977.

5. Dickner, M. E., McGowen, C. L., and Ross, D. T., "Software Design Using SADT," Structured Analysis and Design (Report), Infotech International Limited, Maidenhead, England.

6. Jackson, M. A., "Principles of Program Design," Academic Press, London, 1975.

7. Orr, K. T., "Introducing Structured Systems Design," Structured Analysis and Design (Report), Infotech International Ltd., Maidenhead, England.

8. Yourdon, E. and Constantine, L. L., "Structured Design," Prentice Hall, Englewood Cliffs, NJ, 1979.

9. Gordon, M., "The Byron Program Development Language," Intermetrics Inc., 1982.

10. MacLaren, L., "Evolving Toward Ada in Real Time Systems," ACM Sigplan, Vol. 15, No. 11, November 1980.

11. Jackson, K., "MASCOT and Multiprocessor Systems," Systems Designers Ltd., Surrey, England, February 4, 1983.

12. Prywes, N. S., Pnuelli, A., and Shastry, S., "Use of a Non-Procedural Specification Language and Associated Program Generator in Software Development," ACM Trans. on Programming Languages and Systems, Vol. 1, No. 2, pp 196-217, October 1979.

13. Hamilton, M. and Zeldin, S., "Higher Order Software -- A Methodology for Defining Software," IEEE Trans. on Software Engineering, Vol. SE-2, No. 1, p 9, March 1976.

REFERENCES (continued)

14. Heninger, K. L., Kallander, J. W., Parnas, D. L., and Shore, J. E., "Software Requirements for the A-7E Aircraft," NRL Memorandum Report 3876, Naval Res. Lab., Washington DC 20375, November 27, 1978.

15. Lefkovitz, D. and Hill, H. "The Applicability of Software Development Methodologies to Naval Embedded Computer Systems," Final Report, NADC Contract No. N62269-80-R-0455, University of Pennsylvania, November 9, 1982.

16. U.S. Department of Defense. "Reference Manual for the Ada Programming Language," Berlin: Springer Verlag, 1983.

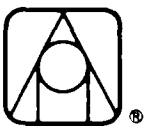
17. Booch, G. "Software Engineering with Ada," Menlo Park, CA: Benjamin/Cummings Publishing Co., 1983.

18. Parnas, D.L., "On the Criteria to be used in Decomposing Systems into Modules," Comm ACM, Vol. 15, No. 12, December 1972.

19. Lefkovitz, D., "An Embedded Software Design Simulator," NADC Contract No. N62269-80-D-0025, June 1984.

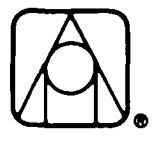
20. Lefkovitz, D., Lee, R., and Nelson, P., "An Embedded Software Design Simulator for Ada Multitasking", Proc. 1985 International Conference on Parallel Processing (ACM), pp 202-207, August 1985.

21. Pritsker, A. D., and Kiviat, P. J., "Simulation with GASP II," Prentice-Hall, Englewood Cliffs, N.J., 1969.



APPENDIX A

SPECIFICATION AND CONCEPTUAL DESIGN OF A VIRTUAL REALTIME MACHINE (VRM)



A. SPECIFICATION AND CONCEPTUAL DESIGN OF A VIRTUAL REALTIME MACHINE (VRM)

A.1 SPECIFICATION OF AN EXPERIMENTAL VRM

Based upon the above principles discussed in Section 2, a VRM can be specified that will perform the following functions:

(1) The user can select one of two queue management types: by FIFO or priority.

(2) The user can preempt the queue.

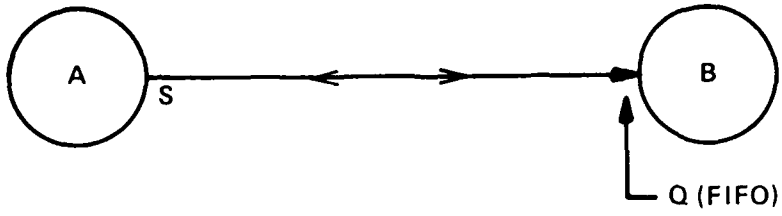
(3) The user can request that the call be executed within a specified time limit.

(4) The call from A to B can either be synchronized or concurrent. If concurrent, B can be requested to call A at completion of service or simply set a flag for A to poll.

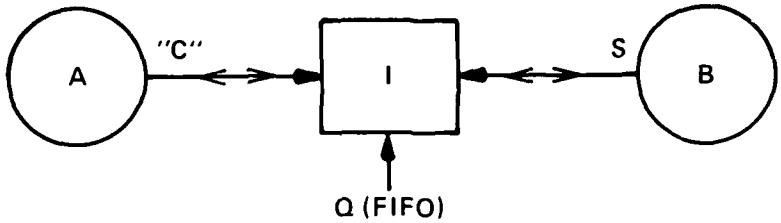
(5) A report will be produced for each task that uses the VRM interface. The type of queue, average queue length, and average wait time over a user-specified time interval will be reported.

Figure A-1 compares the synchronization methods of Ada, MASCOT, and the VRM. In Ada (Figure A-1a), A calls B. It is queued FIFO on B until B services the call. Data parameters are passed in either direction. Until service is complete, A is suspended and hence is said to be fully synchronized (S).

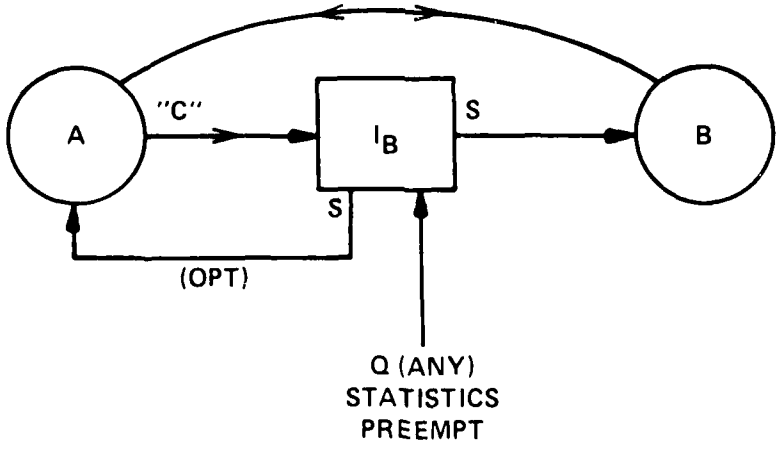
In MASCOT (Figure A-1b), as already described, A calls the interface task I, is queued on I, and is suspended until I services it. In this sense, it is decoupled from B and thus could be viewed as being effectively concurrent with respect to B. Data are buffered in I, and the queues in each direction (A to B and B to A) are maintained within I. In principle, this enables I to service these queues in any desired way; but, in fact, the MASCOT machine is implemented only for FIFO service, probably because the designers saw no need for any other. The types of statistics that may be desirable will be discussed later,



ADA SYNCHRONIZATION
(A-1A)



MASCOT SYNCHRONIZATION
(A-1B)



VRM SYNCHRONIZATION
(A-1C)

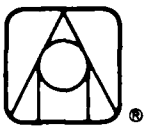
Figure A-1. Three Methods of Task Synchronization (Implementation Diagrams)

but having direct control of the Q enables both MASCOT and the VRM to produce such statistics.

The VRM (A-1C) is a more complex structure. A calls an interface task IB and is released as soon as it is queued by IB. It should be noted, however, that there is also a hidden queue involved in A-1c (namely, Ada's FIFO queue on I itself), since A's call on I is made via the normal Ada invocation of an entry procedure within the task I. The data are passed as parameters in the form of pointers (access-type variables), so data communication can be directly between A and B, eliminating the need for I to be a data buffer. As an additional option, IB could actually buffer data as in MASCOT, but this would preclude the universality of IB, because specific data structures would have to be coded. There does not seem to be any loss of generality in providing for direct data transfer between A and B.

Based upon its own Q management, IB will call B and pass the necessary parameters. For this reason, IB is really associated with B as a front end process, though there is still reason for it to be an independent task and hence be decoupled from B. In the figure, this association is indicated by the notation IB. The Q management could implement any of the service algorithms listed above and can produce Q monitoring statistics at run time. If A wants to continue running after it is queued by IB, then it sets a flag for IB.

It is A's choice after it is queued by IB (i.e., its call is serviced by IB) whether to continue running or to synchronize itself by waiting for a callback from IB, as indicated in A-1c by the call arrow from IB to A. This callback is an option that A specifies in its original call to IB. A could also continue to run and elect the callback option. In the first case, it simply puts an accept entry immediately after its call to IB. In the second case, it puts an accept entry after the processing with which it wants to proceed.



A.2 HIGH LEVEL DESIGN OF THE VRM

In Ada, a program is a procedure that is composed of three kinds of subprogram units: procedures, packages, and tasks. Each of these subunits can be composed of the same three subunits. In the case of a task, the procedure subunit is called an entry if it is visible (i.e., callable) from other tasks. It would be desirable to design the VRM in as universal a manner as possible. This can be done by devising a single, fixed list of parameters that is independent of the data structures that are actually to be passed between tasks, if the functions to be performed by the VRM are indeed common to all tasks. The following design attempts to achieve this:

(1) All entries in the system are to be assigned code values from an enumeration subtype:

```
subtype ENTRY_CODE is INTEGER range 1.. MAXENT
```

(2) A single task, called INTERFACE, will contain the IB procedure of Figure A-2. Symbolically, this will now be called Ij, where j is the ENTRY_CODE with task B (or any other task). A parameter in the call to Ij will indicate which of three queue management disciplines the call wants to use:

(i) FIFO

(ii) Calling task "priority"

(iii) Service time limit

The "priority" in the second case may not be the actual task priority. It is simply a number relative to other numbers that other calling tasks may pass which specifies the priority for the purpose of ordering this queue. The programmer is free to use the actual task priority (or any other number) for this purpose. The service time limit, in the third case, is a time (in pre-established time units) within which the call is to be serviced. The VRM will attempt to adjust its position on the queue in order to satisfy this requirement. If it cannot be satisfied, the call is cancelled.

(3) The INTERFACE task can be implemented either as a family of tasks indexed by an ENTRY_CODE variable or as an access type, where an executive procedure within INTERFACE would dynamically create and terminate subtasks, based upon calls

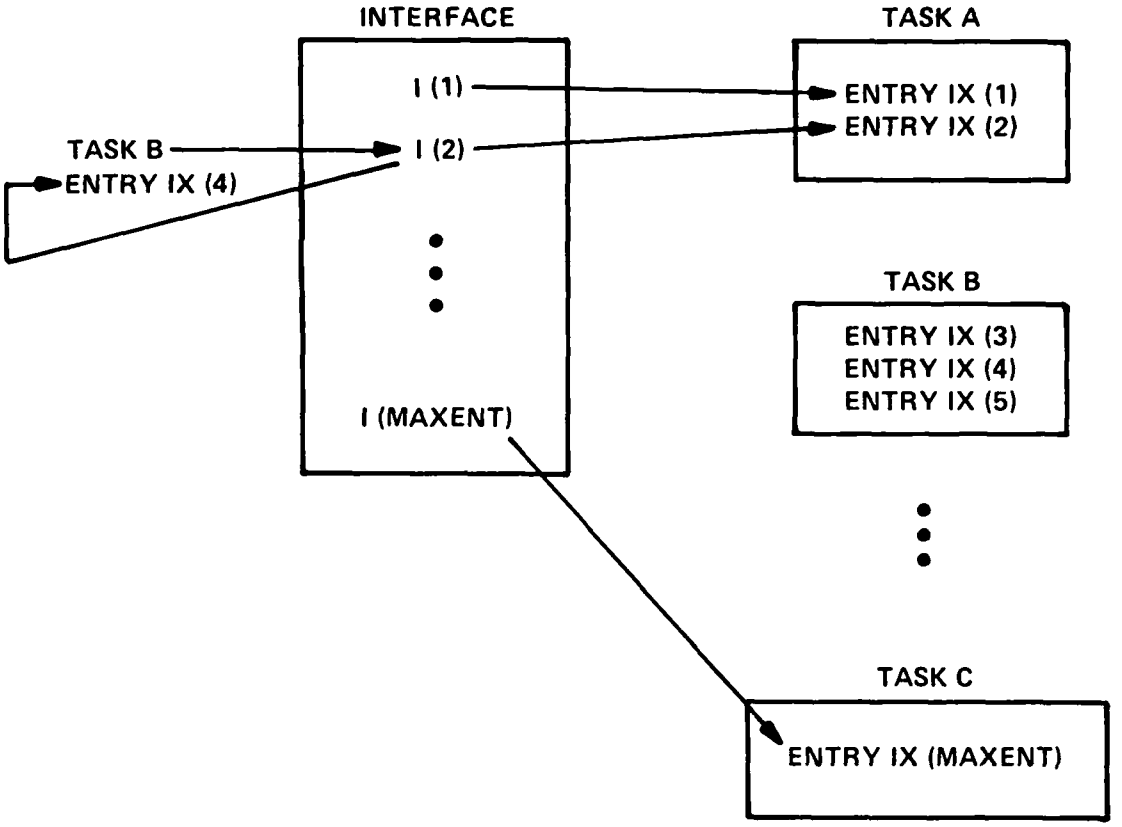


Figure A-2. Schematic of Flow of Control through INTERFACE

to *Ij*. As an example of the former, the general structure of the program would appear as:

```
procedure PROGRAM_X is
     type PARAM is
        record
          -- See Table A-1 for definition
        end record;

     ...
     task INTERFACE is
        entry I (ENTRY_CODE) (P: in PARAM);
     end INTERFACE;
     task body INTERFACE is
     begin
        loop
          select
             accept I(1) (P: in PARAM)
             ...
             end I(1);
          or
```

```
            accept I(2) (P: PARAM)
            ...
            end I(2);
        or
            ...
        end select;
    end loop;
end INTERFACE;
```

(4) Using this approach, Figure A-2 illustrates the control flow through INTERFACE. Task B calls the entry with code 2 (in A). The call is actually made to I(2). Assume that it is a concurrent processing call with callback. I(2) will queue B, if necessary, and at the appropriate time call IX(2), where IX is a family of entries. This is a synchronized call. When complete, I(2) will call IX(4). The synchronized call of I(2) to IX(4) will suspend all of INTERFACE. For this reason, alternative structures that make I(j) separate tasks (such as the access-type approach) should also be investigated.

(5) Table A-1 presents the parameters passed to I(j) in the record variable of type PARAM. The description uses the convention "A calls B". I(j) must move these parameters into a caller queue table associated with each call. There is a table entree for each callee, caller pair. An additional table element, complete, is also needed. It is a Boolean that is initially set to FALSE. When the called task entry IX(j) has completed its processing of the call, complete is set to TRUE. A function COMPLETE (callee, caller) returns the value of complete in the queue table identified by the parameters.

(6) The statistical report is produced using four procedures. INTERVAL (T) sets a time interval, T. A set of statistics is computed every T time units. START_INT starts a statistics-taking interval, and END_INT ends the interval. START_INT is required to start the statistics-taking process. INTERVAL (T) can be called after START_INT to cause a new set to be taken every T time units thereafter and until an END_INT is called. STAT(I_LIST) will print the currently accumulated report containing:

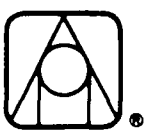
    (i) The average queue length and standard deviation over the recorded interval.

    (ii) The average wait time and standard deviation over the recorded interval.

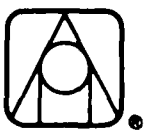
I_LIST may either be a list of integers representing the I(j) queues to be reported or the number "0", which reports all queues. Each queue is identified by its entry code number and the queue type. Also reported are the number of preempts.

Table A-1. CALL Parameters of I

PARAMETER	DESCRIPTION
1. callee	This is the code of the entry being called.
2. caller	This is the code of the entry to be called back, if required by response. It is also used to identify the caller to I(j) for its queue management.
3. response	This is a Boolean that is TRUE if I is to use caller to call A back. If response is FALSE, then no callback is required and caller is ignored.
4. in_param	This is a parameter list of pointers (i.e., access types) to the input parameters within A.
5. out_param	This is a parameter list of pointers that are used by B to store the processing output.
6. queue_type	1 = FIFO 2 = calling task "priority" 3 = service time limit 4 = preempt
7. queue_data	If queue_type = 1, the call is put at the end of the FIFO list and queue_data is ignored. If queue_type = 2 or 3, queue_data contains the relevant value. If queue_ type = 4, the call is put at the head of the FIFO list and queue_data is ignored.



APPENDIX B
FILE STRUCTURES AND PDL OF THE ESDS



B. FILE STRUCTURES AND PDL OF THE ESDS

B.1 ESDS DESIGN DOCUMENTS

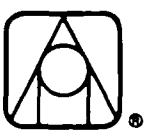
This appendix contains the low level design documentation of the Embedded Software Design Simulator (ESDS). It consists of three documents. Section B2 contains the tables and files that constitute its major data structures. Section B3 contains a call tree of the procedures in the program, and Section B3 contains the PDL.

B.2 DATA STRUCTURES

All of the following structures, whether files or tables, are stored in main memory while the simulator is running. Table B-1 lists and briefly describes the function of each structure.

Table B-1. ESDS Data Structures

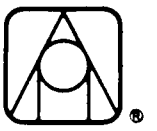
STRUCTURE	DESCRIPTION
Event File	Each event in the simulation is described by a record in the Event File. Table 6-2 in Section 6 is an example of part of the table.
Input Assignment File	The Input Assignment file relates each input type to the event that handles it.
Input File	Every manual input and its time of occurrence appears in the Input file.
Queue File	Each member of a queue is represented by a record in the Queue file.
Time Table	The time of occurrence of automatically generated inputs and delays are stored in the Time Table.
Task Status Table	The status of each task is maintained in this table.



B.2.1 Event File (EV)

DATA ELEMENTS	DESCRIPTION
1. Event-No. (R) *	A unique integer in the Event series.
2. Event-Type (R)	1 = Task initiation 2 = Sequential processing (Entry) 3 = Sequential processing (Non-Entry) 4 = Call (task entry) 5 = End of entry 6 = End of task 7 = Rendezvous (accept, select) 8 = Delay (Non-select) 9 = Delay (select)
3. Exec-Time	Event types 1,2,3,8, and 9 have an associated (For Event-Type = 1,2,3,8,and 9) duration of Exec-Time units. Event types 4 to 7 occur at a moment in time, with no fixed subsequent duration.
4. Output_No (O)	A unique integer in the Output series.
5. Queue-Information (for Event-Type = 4)	
5.1 Queue-Type	1 = FIFO 2 = Calling task priority ** 3 = Service time limit 4 = Preempt **
5.2 Queue-Data (for Queue-Type = 2,3)	(1) Calling task priority for Queue-Type = 2. (2) Service time limit for Queue-Type = 3.
5.3 Synchronization-Type	1 = Unconditional synchronized (rendezvous) 2 = Synchronized with time limit 3 = Synchronized with condition of immediate service 4 = Concurrent with interrupt (call-back) ** 5 = Concurrent with polling ** 6 = Concurrent with no response **
6. Call-Back-Ev-No (O) (for Synchronization-Type = 4)	This is the event number of a type 2 event that represents the call-back entry.

* (R) means that the item is required. (O) means that it is optional.
** Reserved for future VRM implementation.



DATA ELEMENTS (cont'd)	DESCRIPTION (cont'd)
7. Next-Ev-No-1 (O) (List)	This is the event number of the next event, if there is one. In the case of an event type 7, it can be a list of alternative accepts. In the case of event types 1 to 3, it can be a list of alternative events. The alternative is selected by Item 11.
8. Next-Ev-No-2 (O)	There are three control branching situations in the simulation flow. First is a call (Type 4); second is a task initiation (Type 1), and third is a delay (Type 9) that can be interrupted by a call. In these cases, the branched event is *Next-Ev-No-2*, and the next in-line event is *Next-Ev-No-1*.
9. Task List (List) (for Event-Type = 2)	This is the current list of tasks in which this event is active.
10. Priority (R)	This is the priority of the task in which this task occurs.
11. Prob-List (List) (O)	This is a list of probabilities that correspond to the list in *Next-Ev-No-1*.
12. Initiating-Ev-No (R)	This is the initiating event number of the task.

B.2.2 Input Assignment File (IA)

DATA ELEMENTS	DESCRIPTION
1. Input-No (R)	Unique integer in the Input No. series.
2. Event-No (R)	This is the event number that responds to the input.
3. Mode (R)	1 = Task initiation 2 = Interrupt 3 = Polled (For VRM implementation)
4. Output-No (List) (O)	Each of the output numbers that results from the input is listed.
5. Req-I/O-Response (List) (O)	Required response time for each output in Item 4.
6. Min-I/O Response (List) (O)	Computed minimum response time for each output in Item 4.

B.2.3 Input File (INPUT)

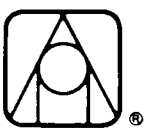
DATA ELEMENTS	DESCRIPTION
1. Input-No. (R)	Unique integer in the Input series.
2. Event-Time (R)	The time at which the input is to occur.

B.2.4 Queue File (Q)

DATA ELEMENTS	DESCRIPTION
1. Calling-Task-No. (R)	Task number of the calling event.
2. Called-Event-No. (R)	The entry that is called.
3. Calling-Event-No. (R)	The event number of the calling event.
4. Queue-Type (R)	See Item 5.1 in the EV File for types.
5. Call-Time (R)	The time at which the call was made.
6. Queue-Time (O)	In the case of a time condition, this is the queue cancel time.
7. Calling-Task-Priority (R)	Priority of the calling task.

B.2.5 Time Table (TT)

DATA ELEMENTS	DESCRIPTION
1. Time (R)	Time at which automatically generated input is to occur or delay is to expire.
2. Type (R)	1 = Automatic input 2 = Delay
3. Input-No. (O)	Input No. if Type = 1.
4. Event-No. (O)	Event No. if Type = 2.
5. Regeneration-Flag (O)	This is a number that designates an algorithm for generating the next input time. If the flag is 0, there is no regeneration.
6. Task-No. (O)	This is the task number, if Type = 2.



B.2.6 Task Status Table (TST)

DATA ELEMENTS	DESCRIPTION
1. Task-No. (R)	Task number of table entry.
2. Priority (R)	Current task priority.
3. Current-Ev-No. (R)	Current event number.
4. Status (R)	1 = Suspended (delay) 2 = Suspended (rendezvous) 3 = Suspended (call wait) 4 = Running 5 = Executing
5. First-S-T (R)	First execution start time of the current event.
6. Last-S-T (R)	Last execution start time of the current event.
7. Time-Rem (R)	Time remaining in the execution of the current event.
8. Calling-Ev-No. (O)	Calling event number.
9. Calling-Task-No. (O)	Task No. of the calling event.
10. Time-Stat-On (R)	The time at which the Status was changed.
11. Initiating-Ev_No. (R)	Initiating event number of the task.
12. PT (O)	Parent task number.
13. CT (O)	Child task number.
14. ST (O)	Sibling task number.
15. Terminate-Flag	Y = Task is eligible for termination. N = Task is not eligible for termination.
16. Select-Terminate	This flag is set when a type 7 event is processed and there exists a *terminate* command in the *select*.

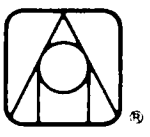
B.3 CALL TREE OF THE PDL

Figure B-1 presents the call tree of the major procedures in the ESDS PDL. The PDL itself is given in Section B4. This type of diagram has certain conventions that indicate some of the procedure call logic. An open circle with a number over it indicates an iteration of calls in a sequence indicated by the numbers. A filled-in circle indicates a conditional choice of one of the subtending procedures. No circle means that all of the subtending procedures are called but not necessarily in any indicated sequence and not in an iteration.

B.4 PDL OF THE ESDS

The following conventions are used in this PDL:

```
-- is a comment.

-- & is a pseudocommand.  It appears within the normal program's
   syntactic structure, but it will contain natural language
   specifications.

   --Event Types
     --1= Task initiation
     --2= Sequential processing (entry)
     --3= Sequential processing (non-entry)
     --4= Call  (Task entry)
     --5= End of entry
     --6= End of task
     --7= Rendezvous (accept or select)
     --8= Delay  (non-select)
     --9= Delay  (select)

   --Status codes
     --1= Suspended  (Delay)
     --2= Suspended (Rendezvous)
     --3= Suspended (Call wait)
     --4= Running
     --5= Executing
```

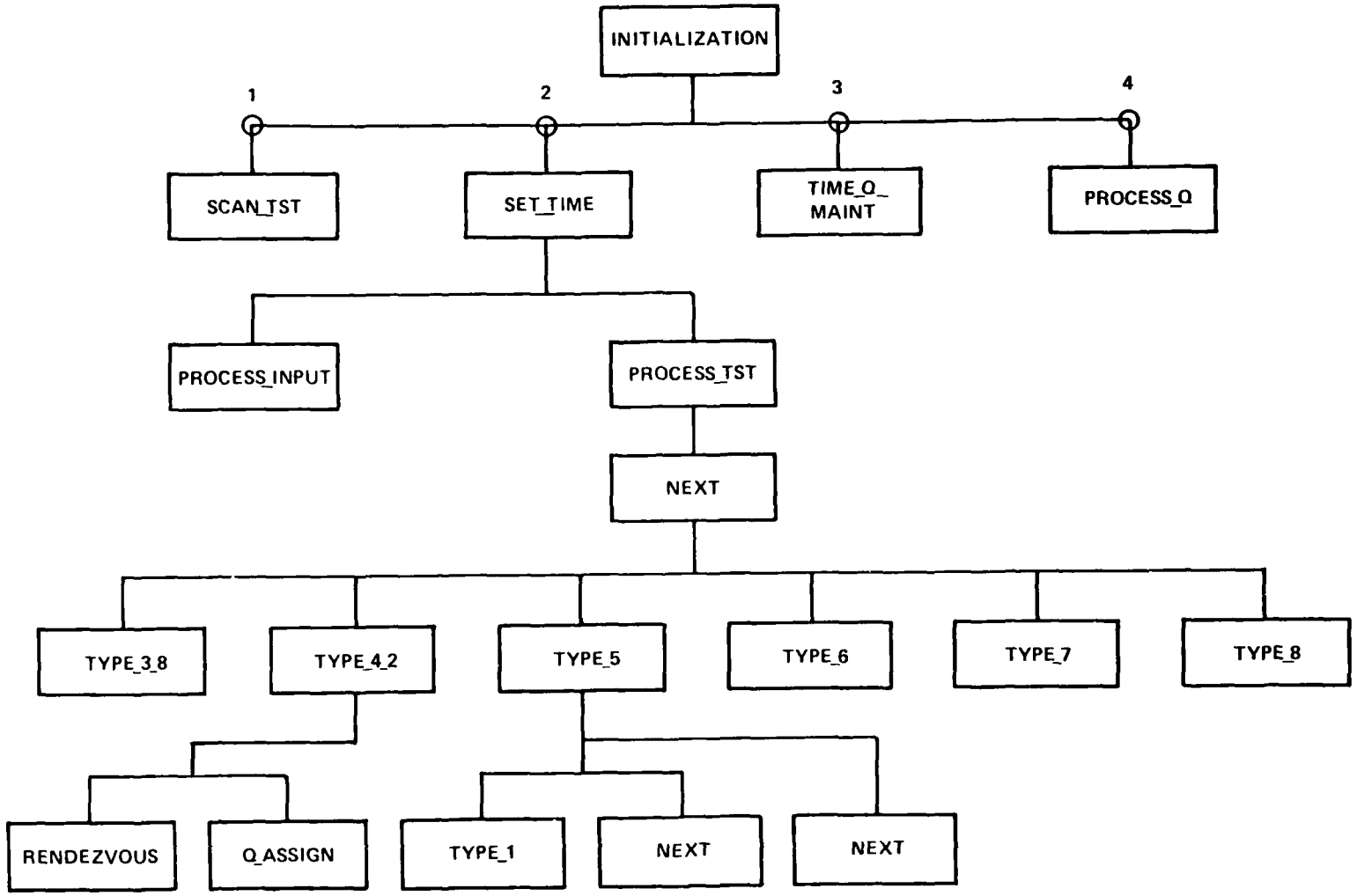
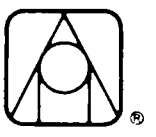
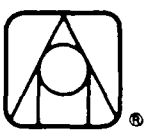


Figure B-1. Call Tree of the ESDS PDL

```
procedure INITIALIZATION is
begin
              -- & Initialize global variables: I_NO, TN, TND<E_TST, E_TT,
                   T_TABLE, T_TST
              -- & Initialize statistics;
              -- & Initialize TST
              TIME:= 0;
                     if AUTO_GEN then
              loop  I  in 1..MAX_GEN_NO
                    GENERATE(I,Time(TT),Input-No(TT));
                    Regeneration -Flag(TT):= I;
                    TYPE(TT):= 1:
              -- & Add record to Time Table based on value of Time (TT);
                 end loop;
     end if;
     NEXT_TIME;
end INITIALIZATION;
procedure NEXT_TIME is

              -- All inputs and delays are entered into the Time Table
              -- in ascending order by scheduled time of occurrence for
              -- inputs and time of expiration for delays. NEXT_Time reads
              -- the next record in the Time Table and assigns the time
              -- to the variable T_TABLE.
begin
              -- & Read next Time Table (TT) record;
                   T_TABLE:= Time (TT);
                   case Type (TT) is
                   when 1 => I_NO:= Input-No(TT),
                   when 2 =>
                   TND:= Task-No(TT);
                   E_TT:= Event-No(TT);
                   I_NO:= 0;
     end case;
     if Regeneration-Flag(TT) /= 0 then
        GENERATE(Regeneration-Flag(TT),Time(TT),Input-No(TT));
              -- & Add record to Time Table based on value of Time (TT);

     end if;
     end NEXT_TIME;

procedure SCAN_TST is

              -- This procedure selects the next event in the TST to be
              -- completed.  The selection criteria are hierarchically applied
              -- as: (1)Maximum priority, (2)task is executing (Status=5) or
              -- running (Status=4), and (3) earliest event start time. The
              -- selected task is started (or continued) in execution (Status =
              -- 5) at time (the current time) and is tentatively scheduled for
              -- completion at time T_TST = TIME + Time-Rem(TST).
```

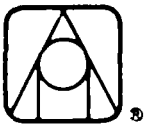



```
begin
          FOUND_TST:= FALSE;

               -- & Find the TST record (TN) with MAX(Priority (TST)) and Status (TST) =
                  4:5;
               -- & If one or more records (TN) found then
          FOUND_TST:= TRUE:

               -- & If Status(TST)=5 for any of the found records then
               -- & Find a record (TN) with MIN(First-S-T) and then MIN(Time-
                    Rem(TST));
                    T_TST:=  [Last-S-T(TST)] + [Time-Rem(TST)];
else           -- Status(TST) = 4
               -- & Find a TST record (TN) with MN(First-S-T(TST)) and then
                    MIN (Time-Rem (TST));
                    STAT_TASK(TN);
                    Status(TST):= 5;
                    T_TST:= TIME + [Time-Rem(TST)];
                    Last-S-T(TST):= TIME;
                    if MULTI-PROC then MULTI_PROCESS;

                   else loop

               -- & For every TST record(X /= TN):
                  if Status(TST(X) = 5 then
                  STAT_TASK(X);
                  Status (TST(X)):=4;
                  TIME-REM(TST(X)):= Time-Rem(TST(X)) -
                  (TIME - [Last-S-T(TST(X))]);
end if;
          end loop;
          end if;
          end if;
          end if;

          if not FOUND_TST then
               -- & Print Message: "Entire system is inactive or suspended at an
               -- entry rendezvous."
           T_TST:= HI;
                return;
else
          E_TST:= Cur-Ev-No(TST);
          end SCAN_TST;


procedure SET_TIME is
               -- This procedure selects an input, delay or TST event as the
               -- next to actually complete, depending upon which occurs
               -- first. The time of this input, delay or event is assigned to
               -- TIME> In the case of an input, PROCESS_INPUT is called.
               -- In the case of a delay or TST event, PROCESS_TST is called.
```

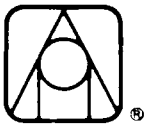



```
begin
              TX:= T_TABLE;
              if T_TABLE <= T_TST then
                   TIME:= T_TABLE;
              if I_NO /= 0 then PROCESS_INPUT;          -- it's an input
              else                                      -- it's a delay
              PROCESS-TST(E_TT,TND);
              -- & if Status(TST(X))=5 for any task X and Priority(TST(TND))
              -- > Priority(TST(X)) then
                   STAT_TASK(X);
                   Status(TST(X)):+ 4;
                   Time-Rem(TST(X)):= [Time-Rem(TST(X))] - (TIME -
                   [Last-S-T(TST(X))]);
              --   end if;
          end if;
          NEXT_TIME;
end if;
          if T_TST <= TX then
          TIME:= T_TST;
          PROCESS_TST(E_TST,TN);
end if;
          end SET_TIME;

          procedure PROCESS-INPUT is

               -- There are three input modes. Mode 1 is an artificial one that
               -- initiates a task. It really represents a task elaboration
               -- within the structure of the Ada program and hence should be
               -- given an event time of zero. Initiation of task objects within
               -- the program are handled by a specific event of type 1. Mode 2
               -- is an interrupt and is handled, as in Ada, as a high priority
               -- task entry. Mode 3 is a polled input. Initially, it will not
               -- be implemented. In the processing, a Mode 1 input assumes an
               -- inactive(Status=1) task. Mode 2 will run immediately, with
               -- high priority if the task in inactive or at a rendezvous;
               -- otherwise, it is queued on the high priority INTERRUPT queue.
begin
          EX:= Event-No(IA(I-NO));
               when 1 =>
                    TNX:= NEWTASK;
                    Priority(TST(TNX)):=Priority(EV(EX));
                    Cur-EV-No(TST(TNX)):= EX;
                    Initiating-Ev-No(TST(TNX)):= EX;
                    INIT-LINKS(TNX);
               when 2 =>
                    TASK-RANDOM(EX,TNX);
                    RENDEZVOUS(EX,TNX,FLAG);
                    if FLAG then Priority (TST(TNX)):= HI;
                    else
                         Q_ASSIGN(EX,TNX,0,0,TRUE);
                         return;
```

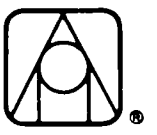

```
                    end if;
               when 3 =>
                    ERROR (2);
                    return;
end case;
               -- Load TST record for TNX:
                    Status(TST(TNX)):= 4
                    First-S-T(TST(TNX)):= TIME;
                    Time-Rem(TST(TNX)):=Exec-Time(EV(EX));
                    Time-Status-On(TST(TNX)):= TIME;
               -- & If Output-No(1) (IA(I_NO))  /= 0 then add a record (J) for each
                    Output-No(I) to the IOT:
                         Input-No(IOT(J)):= I_NO;
                         Output-No(IOT(J)): Output-No(I) (IA(I-NO));
                         Input-Time(IOT(J)):= TIME;
                    end if;

end PROCESS_INPUT;
         INIT_LINKS(TX:in) is
begin
         PT(TST(TX)):= 0;
         CT(TST(TX)):= 0;
         ST(TST(TX)):= 0;
         Terminate-Flag (TST(TX)):= "N";
end INIT_LINKS;

ADD_TASK (PARENT:in CHILD:in) is
begin
         if CT (TST(PARENT)) =0 then
              CT (TST(PARENT)):= CHILD;
         else
              -- & Chain from CT(TST(PARENT)) to ST(TST(X1)) through ST(TST(Xn)),
              -- where Xi is a sibling task and ST(TST(Xn)) = 0;
          ST(TST(Xn):= CHILD;
                    end if;
           PT (TST(CHILD)):= PARENT;
end ADD_TASK;


TERM_TASK (TX:in) is
begin
         Terminate-Flag (TST(TX)):= "Y";
              -- & If the Terminate-Flag = "Y" for every task in the entire tree
                 in
              -- which TX is embedded, then terminate the entire tree by removing
              -- the corresponding records from the TST.
end TERM_TASK;

procedure RENDEZVOUS (ex6:in; TX6:in; Flag:out) is
              -- This procedure determines whether task TX6 is at a
              -- rendezvous at entry event EX6.  If it is, FLAG:= TRUE;
              -- if not, FLAG:= FALSE.
```

```
begin
         if Status(TST(TX6)) = 2 and LIST(Cur-Ev-No(TST(TX6)),EX6) then
              FLAG:= TRUE;
         else if Status(TST(TX6)) = 1 and LIST(Next-EV-No-1(EV(Cur-EV-No(TST(
              TX6))))),EX6) then
                    FLAG:= TRUE:
              -- & Find and delete Time Table record with Task-No(TT) = TX6;
         else FLAG: = False;
         end if;
   end case;
end RENDEZVOUS;

function LIST (ET7:in; ECOMP:in) return BOOLEAN is
              -- This function looks for an open accept at the select (ET7)
              -- for the entry ECOMP. The function returns TRUE if it is
              -- found; otherwise, it returns FALSE.
begin
         if Prob-List (1) (EV(ET7)) = 0 then
              -- & if any Next-Ev-No-1(i)(EV(ET7)) = ECOMP then return TRUE;
              -- else return false;
              -- end if;
              else
                  RAND(SEED,R);
              -- & if any Next-Ev-No-1(i)(EV(ET7)) = ECOMP and Prob-List(i)
              -- (EV(ET7)) >= R then return TRUE;
              -- else return FALSE;
          end if;
end LIST;


procedure TASK-RANDOM (EX7:in; TX7:out) is

              -- This procedure selects at random a task in which EX7
              -- is currently contained.

begin
              -- & Select a task, TX7, at random from among the tasks in
              -- Cur-Task-Nos(EV(EX7));
end TASK_RANDOM;


procedure PROCESS_TST (E1:in; TN1:in) is

              -- This procedure is reached when an event has just completed.
              -- PROCESS_TST will find the next event (or events) to be run and
              -- establish them in the TST. Three conditions can establish two
              -- (branching) events. The first is a task initiation by a task and
              -- the continuation of the initiating task. Second is an entry com-
              -- pletion, which causes a return event to the calling task to be
              -- established and either a return to rendezvous or task completion
              -- in the current task. Third is the interruption of a delay within
```

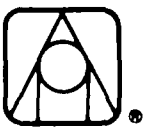



```
          -- the scope of a select, which branches to an accept. If the next
          -- event is a Call to a running task, then the call is queued. Note
          -- that this procedure has a case selection only to Event-Types 1,2,
          -- 3, and 4, because these are the four with non-zero execution time.
          -- The other four types are processed within the selection of the
          -- first four as "pass-throughs" (under procedure NEXT), and they
          -- will never show up in SET_TIME for time incrementing or statistics
begin
        if Output-No(EV(E1)) /= ) then STAT_INPT(Output-No(EV(E1)));
        end if;
        STAT_EV(E1,TN1);
        X1:= Next-Ev-No-1(EV(E1));
        X2:= Next-Ev-No-2(EV(E1));
        if X1 = 0 then ERROR(3);
        else case Event-Type(EV(E1)) is
                when 1 =>
                     if X2 = 0 then
                         if Event-Type(EV(X1)) = 3| 4| 6| 7| 8 then NEXT(X1,TN1);
                         else ERROR(14);
                         end if;
                     else if Event-Type(EV(X2)) = 1 then
                          if Event-Type(EV(X1)) = 3| 4| 6| 7| 8 then
                                TYPE_1(X2,E1,TN1);
                                NEXT(X1,TN1);
                     else ERROR(7);
                     end if;
                   else ERROR(8);
                   end if;
                when 2 =>
                     if X2 = 0 then
                         if Event-Type(EV(X1)) = 3| 4| 5| 8 then NEXT(X1,TN1);
                         else ERROR(15);
                         end if;
                     else if Event-Type(EV(X2)) = 1 then
                          if Event=Type(EV(X1)) = 3| 4| 5| 8 then
                                TYPE_1(X2,E1,TN1);
                                NEXT(X1,TN1);
                     else ERROR(4);
                     else ERROR (5);
                     end if;
                when 3| 8 =>
                          if X2 = 0 then
                       if Event-Type(EV(X1))= 3| 4| 5| 6| 7| 8 then NEXT(X1,TN1);
                       else ERROR(15);
                       end if;
                  else if Event-Type(EV(X2)) = 1 then
                       if Event-Type(EV(X1)) = 3| 4| 5| 6| 7| 8 then
                            TYPE_1(X2,E1,TN1);
                            NEXT(X1,TN1);
                       else ERROR(6);
                       end if;
```

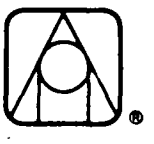
```
                    else ERROR(11);
                    end if;
               when 9 =>
                    if Event-Type(EV(X1) = 3| 4| 6| 7| 8 then NEXT(X1,TN1);
                    else ERROR(12);
                    end if;
end case;
               end if;

end PROCESS_TST;

procedure NEXT(EX:in; TX:in) is

               -- NEXT is passed as a parameter the next event to be run. It
               -- either loads a TST entry or modifies the Status in an existing
               -- one, using a series of procedures according to Event Type.
               -- The branching condition of an end of entry (Event Type 5) is
               -- handled within NEXT (case 5) and TYPE_5.
begin
     case Event-Type(EV(EX)) is
               when 3 => TYPE_3_8(EX,TX,4);
               when 4 => TYPE_4_2(EX,TX);
               when 5 =>
                    N1:= Next-Ev-No-1(EV(EX));
                    N2:= Next-Ev-No-2(Ev(EX));
                    TYPE_5(N1,N2,TX);
                    Priority(TST(TX)):= Priority(EV(Cur-Ev-No(TST(TX))));
                    if Calling-Ev-No(TST(TX)) = 0 then return;
                    else
                       X:= Calling-Ev-No(TST(TX));
                       EX:= Next-Ev-No-2(EV(X));
                       TX:= Calling-Task-No(TST(TX));
                       NEXT(EX,TX);
               end if;
          when 6 => TYPE_6(TX);
          when 7 => TYPE_7(EX,TX);
          when 8 => TYPE_3_8(EX,TX,1);
          when 1| 2| 9 => ERROR(13);
    end case;
end NEXT:
procedure TYPE_1(EX1:in; EY1:in; TY1:in) is

               -- This procedure assigns a new task number and loads a TST entry.
               -- EX1 is the type 1 event to be initiated; EY1 is the event that is
               -- initiating it, and TY1 is the task that is initiating it.
begin

          TNX:= NEWTASK;
               if Task-Object(EV(EY1)) = "Y" then Task-Object(TST(TYI)):= TNX;
               else Task-Object(TST(TY1)): 0;
end if;
```


```
                -- Load TST record for TNX:
                        Cur-Ev-No(TST(TNX)):= EX1;
                        Initiating-Ev-No(TST(TNX)):= EX1;
                        Status(TST(TNX)):= 4;
                        First-S-T(TST(TNX)):= TIME;
                        Time-Rem(TST(TNX)):= Exec-Time(EX1));
                        Priority(TST(TNX)):= Priority(EV(EX1));
                        INIT_LINKS(TNX);
                        ADD_TASK(TY1,TNX);
end TYPE_1;

procedure TYPE_3_8(EX3:in; TX3:in; STAT:in) is

                -- This procedure loads a TST entry for an event type 3
                -- (sequential processing - non-entry) or type 8 (delay)
begin
     STAT_TASK(TX3);
                -- Load TST record for TX3:
                   Cur-Ev-No(TST(TX3)):= EX3;
                   Status(TST(TX3)):= STAT;
                   First-S-T(TST(TX3)):= TIME;
                   Time-Rem(TST(TX3)):= Exec-Time(EV(EX3));
                Time(TT):= Exec-Time(EX(EX3));
                Type(TT):= 2;
                Task-No(TT):= TX3;
                -- & Add record to Time Table based on value of Time (TT);
end TYPE_3;

procedure TYPE_4_2(EX2:in; TX2:in) is

                -- This procedure suspends the calling task(Type 4). If the called
                -- task is at the rendezvous, it loads the TST with the called
                -- entry.  If the called task is not at the rendezvous, it queues
                -- the call.
begin

     STAT_TASK(TX2);
                -- Suspend Calling Task:
                   Status(TST(TX2)):=3;
                   Called_EVENT:= Next-Ev-No-1(EV(EX2));
                   if Task-Object(TST(TX2)) = 0 then TASK_RANDOM(CALLED_
                   EVENT,CALLED_TASK);
                   else CALLED_TASK:= TasObject(TST(TX2);
                   end if;
                   RENDEZVOUS(CALLED_EVENT,CALLED_TASK,FLAG);
                   if FLAG then
                   if Priority(TST(TX2)) >Priority(TST(CALLED_TASK)) then
                      Priority(TST(CALLED_TASK)):= Priority(TST(TX2));
                   end if;
                   STAT_TASK(CALLED_TASK);
                   Status(TST(CALLED_TASK)):= 4;
```

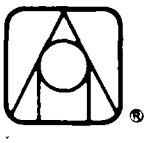
```
                    First-S-T(TST(CALLED_TASK)):= TIME;
                    Time-Rem(TST(CALLED_TASK)):= Exec-Time(EV(CALLED_EVENT));
                    Calling-Ev-No(TST(CALLED_TASK)):= EX2;
                    Calling_Task-No(TST(CALLED_TASK):= TX2;
    else
                    Q_ASSIGN(CALLED_EVENT,CALLED_TASK,EX2,TX2,FALSE);
    end if;
end TYPE_4_2;


procedure TYPE_5(NX1:in; NX2:in; TX1:in) is

              -- This procedure processes the next event of the called task
              -- after end of entry. The next event of the calling task is
              -- handled back in NEXT.

begin
         if NX2 = 0 then
         if Event-Type(EV(NX1)) = 3| 4| 6| 7| 8 then NEXT(NX1,TX1);
         else ERROR(17):
         end if;
         else if Event-Type(EV(NX2) = 1 then
              if Event-Type (EV(NX1)) = 3| 4| 6| 7| 8 then
              TYPE_1(NX2,EX,TX1);
              NEXT(NX1,TX1):
         else ERROR(18);
         end if;
     else ERROR(19);
     end if;
end TYPE_5;

procedure TYPE_6(TX4:in) is

              -- This procedure terminates a task.

begin
         STAT_TASK(TX4);
         No-Instances(TASK_STAT(Initiating-Ev-No(TST(TX4)):= [No-Instances(TASK_
         STAT(Initiating-Ev-No(TST(TX4))] +1;
         TERM_TASK(TX4);
end TYPE_6;

procedure TYPE_7 (EX5:in; TX5:in) is

              -- This procedure corresponds to a select or accept statement. It
              -- sets Status = 2 for a task that reaches a rendezvous (accept
              -- statement) or Status = 1 for a delay.
begin
     STAT_TASK(TX5);
     FOUND:= FALSE;
               if Prob-List(1) (EV(EX5)) =0 then
```


```
                -- & for every X = Next-Ev-No-1(i)(Ev(EX5)) loop
                -- See if there is a queued call
                -- & if Q(TX5,X) exists then
                     SERVE_Q(TX5,X);
                     FOUND:= TRUE;
                       exit;
                -- & end if;
                -- & end loop;
           else
                RAND(SEED,R);
               -- & for every X = Next-Ev-No-1(i)(EV(EX5)) and
               -- Prob-List(i)(EV(EX5)) >= R loop
               -- See if there is a queued call to an
               -- open entry
               -- & if Q(TX5,X) exists then
                  SERVE_Q(TX5,X);            --Call exists
                  FOUND:= TRUE;
           exit;
                -- end if;
                -- end loop;
           end if;
                if not FOUND then                              -- No call exists
                     if Next-Ev-No-2(EV(EX5)) = 0 then         -- No delay or else
                                                               -- exists
                        Status(TST(TX500:= 2;
                        Cur-Ev-No(TST(TX5)):= EX5;
           else                                                -- delay or else exists
                        if Event-Type(EV(Next-Ev-No-2(1)(EV(EX5)) /= 9 then
                                                               -- else exists
                        NEXT(Next-Ev-No-2(EV(EX5)),TX5);
           else
                             Status(TST(TX5):= 1;
                             First-S-T(TST(TX5)):= TIME;
               -- & Find x,Y, where X= min Exec-Time(EV(Y(I))),
               -- Y(I) = Next-Ev-No-2(i)(EV(EX5)))) for all i,
               -- and I is the value of i for which Exec-Time
               -- is a minimum;
                  Time-Rem(TST(TX5)):= X;
                  Cur-EV-NO(TST(TX5)):= Y(I);
                  Time(TT):= X;
                  Type(TT):= 2;
                  Task-No(TT):= TX5;
              -- & Add record to Time Table based on value
              -- of Time(TT);
         end if;
    end if;

end TYPE_7;

procedure Q_ASSIGN (ED:in; TD:in; EG:un; TG:in; INT:in) is
              -- This procedure assigns an entry to its appropriate queue by
```

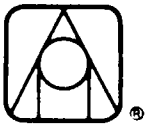



```
                -- Calling and Called entry event numbers and by
                -- Queue-TYPE. Note that the INTERRUPT queue is also assigned
                -- here (See PROCESS_INPUT). ED is the called event. TD is the
                -- called task. EG is the calling event. TG is the calling task.
                -- INT is the INTERRUPT queue flag.

begin
                -- & Create a new Queue record (Q):
                     Called-Task(Q):= TD;
                     Called-Ev-No(Q):= ED;
                     Calling-Task-No(Q):= EG;
          if INT then
                     Queue-Type(Q):= 5;
                     Calling-Task-Pri(Q):= HI;
   else
                Queue-Type(Q):= Queue-Type(EV(EG));
                Queue-Type(Q):= Queue-Data(EV(EG));
                Calling-Task-Pri(Q):= Priority(TST(TG));
          end if;
                Call-Time(Q):= TIME;
                case Queue-Type(Q) is
                     when 1 =>
                -- & Add to FIFO Queue
                     when 2 =>
                -- & Add to PRIORITY Queue using Queue-Data(Q);
                     when 3 =>
                -- & Add to TIME Queue using Queue-Data(Q);
                     when 4 =>
                 --& Add to top of HIGH PRIORITY Queue;
                     when 5 =>
                -- & Add to INTERRUPT Queue;
   end case;
end Q_ASSIGN;

procedure TIME_Q_MAINT is

                -- This procedure examines each Time queue. If the current time
                -- (TIME) is greater than the scheduled event time of any call
                -- on the queue, then the entry call is cancelled.

begin
     loop
                -- & For the HOL(Q) of each TIME Queue:
                     if Queue-Data(Q) < TIME then
                      STAT_Q(Q);
                -- & Delete record Q from TIME Queue;
                     end if;
     end loop;
end TIME_Q_MAINT;
```


```
procedure PROCESS_Q is

              -- This procedure examines each queue, in queue priority order,
              -- where the highest priority is that of the INTERRUPT queue, and
              -- the rest are specified as a parameter within the BUILD_MODEL
              -- program. A TST entry is loaded for any task that is at a
              -- rendezvous.

begin
              -- & For HOL(Q) of each Queue (in Queue Priority Order), where Q is
              -- defined as (Task-No, Called-Ev-No), loop
                 RENDEZVOUS(Called-Ev-No(Q), Task-No(Q),FLAG);
                 if FLAG then serve_Q(Task-No(Q),Called-Ev-No(Q));
                 end if;
              -- & end loop;
end PROCESS_Q;


procedure SERVE_Q(TNX:in; EX:in) is

              -- Q is defined by the Key:
              -- TNX
              -- EX
begin
         STAT_TASK(TN);
              if Calling-Task-Pri(Q) > Priority(TST(TNX)) then
                 Priority(TST(TNX):=Calling-Task-Pri(Q);
          end if;
                 Cur-Ev-No(TST(TNX):= EX;
                 Status(TST(TNX)):= 4;
                 First-S-T(TST(TNX)):= TIME;
                 Time-Rem(TST(TNX)):= Exec-Time(EV(EX));
                 Calling-Ev-No(TST(TNX)):= Calling-Ev-No(Q);
                 Calling-Task-No(TST(TNX)):= Calling-Task-No(Q);
                 STAT_Q(Q);
              -- & Delete record Q from Queue;
end SERVE_Q;
procedure STAT_INPT(TPX:in) is
begin
              -- & Find the IOT record (I) with Output-No(IOT(I) = TPX and
              -- MIN(Input-Time(IOT(I));
              -- & If none found then ERROR(9);
         else
                 X:= Input-No(IOT));
                 Y:= Output-No(IOT));
                 No-of-10-Occurences(IN_OUT_STAT(X,Y)):= [No-of-10-Occurences
                 (IN_OUT_STAT(X,Y)) + 1;
         Total-10-Resp-Time(IN_OUT_STAT(X<Y)):= [Total-10-Resp-Time
         (IN_OUT_STAT(X,Y))] + TIME - [Input-Time(IOT(I))];
              -- & Release record I from IOT;
end STAT_INPT;
```

```
procedure STAT_EV(EX:in; TX:in) is
begin
          Total-Run-Time(EV_STAT(EX)):= [Total-Run-Time(EV_STAT(EX))] + TIME -
           [First-S-T(TST(TX))];
          Total-Exec-Time(EV_STAT(EX)):= [Total-Exec-Time(EV_STAT(EX))] +
           [Exec-Time(EV(EX))];
end STAT_EV;

procedure STAT_Q(Q:in) is
begin
               -- & Add a record to the Q-HIST file:
                  Queue-Type(Q_HIST):= Queue-Type(Q);
                  Called-Ev-No(Q_HIST):= CalledEv-No(Q);
                  Q-Start_Time(Q_HIST):= Call-Time(Q);
                  Q-End-Time(Q-HIST):= TIME;
end STAT_Q;

procedure STAT_TASK(TX:in) is
begin
         EX:= Initiating-EV_NO(TST(TX));
              case Status(TST(TX)) is
                   when 1 =>
              Suspend-Delay-Time(TASK_STAT(EX)):= [Suspend-Delay-Time
              (TASK_STAT(EX))] + TIME - [Time-Status-One(TST(TX))];
                   when 2 =>
              Suspend-Rend-Time(TASK_STAT(EX):= [Suspend-Rend-Time
                 (TAS_STAT(EX))] + TIME - Time-STATUS_On(TST(TX))];
                   when 3 =>
              Suspend-Call-Time(TASK_STAT(EX)):= [Suspen-Call-Time
              (TASK_STAT(EX))] + TIME - [Time-STATUS_ON(TST(TX))];
                   when 4 =>
              Run-Time(TASK_STAT(EX)):= [Run-Time(TASK_STAT(EX))]
              + TIME - [Time-Status-ON(TST(TX))];
                   when 5 =>
              Exec-Time(TASK_STAT(EX)):= [Exec-Time(TASK_STAT(EX))]
              + TIME - [Time-Status-On(TST(TX))];
     end case;
              Time-Status-On(TST(TX)):= TIME;
end STAT_TASK;

         procedure MULTI_PROCESS is separate;
         procedure ERROR(CODE:in) is separate;
         procedure GENERATE(ALGO_NO:IN; INPUT_TIME:out; INPUT_NO:out) is
         separate; function NEWTASK return TASK_NUMBER_TYPE is separate;


------- Main Program --------

         procedure ESDS is
              -- & Global variable declarations;
begin
```

```
        INITIALIZATION
   loop
        SCAN_TST;
        SET_TIME;
        TIME_Q_MAINT;
        PROCESS_Q:
    end loop;
end ESDS
```

B.4.1 EVENT TRANSITION TABLE

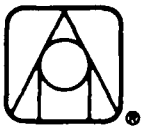
Table B-2 specifies all of the possible inter-event transitions.

Table B-2. Event Transition Table

EVENT TYPE	NEXT-1*	NEXT-2*
1 Task Initiation	F(3,4,6,7,8)	0 or 0(1)
2 Seq Processing (entry)	F(3,4,5,8)	0 or 0(1)
3 Seq Processing (non-entry)	F(3,4,5,6,7,8)	0 or 0(1)
4 Call	F(3,4,6,7,8)	0(2)
5 End of Entry	F(3,4,6,7,8)	0 or 0(1)
6 End of Task	0	0
7 Rendezvous (accept, select)	F(List of 2s)	0 or F(3,4,6,8) or F(List of 9s)
8 Delay (non select)	F(3,4,5,6,7,8)	0 or 0(1)
9 Delay (select)	F(3,4,6,7,8)	0(7)

* F = Forward control
  0 = Out (Branching) control

APPENDIX C
ESDS INTERACTIVE DISPLAY SPECIFICATION



C. ESDS INTERACTIVE DISPLAY SPECIFICATION

C.1 INTERACTIVE DISPLAYS

There are five interactive displays (reports), from summary to detail. The five report specifications are presented in Section C2 to C6. They are specified in four parts: First is a HEADER, which is the information displayed at the top of the report. Second are SELECTION criteria for producing the report. Third is a format specification for the HORIZONTAL dimension of the report. These are the column headings. Fourth is the format specification for the VERTICAL dimension. These are the detail lines.

C.2 TASK SUMMARY

HEADER:

1. Selection
2. Time

SELECTION:

1. All Tasks
2. Selected Tasks
3. Selected Stati

HORIZONTAL:

TASK

1 - - - n

VERTICAL:

1. Status
2. Priority
3. Task Efficiency-1 [= EX/(RN + EX)]
4. Task Efficiency-2 [= (RN + EX)/(RN + EX + SD + SR + SC)]
5. Initial Event No.
6. Task Object [Y/N]
7. Initiating Task [If Task Object]
8. No. of Queues Active
9. Avg. Queue Size

10. Inputs Pending [In queue and in process]
11. Avg. I/O Efficiency
12. No. of I/O Failures

C.3 TASK DETAIL - I/O SUMMARY

HEADER:

1. Task No.
2. Time
3. DISPLAY NO. 1 Detail

SELECTION:

1. None

HORIZONTAL:

INPUT/OUTPUT PAIR

1 - - - n

VERTICAL:

1. Input No.
2. Input Event No.
3. Output No.
4. Output Event No.
5. Required Response Time
6. Minimum Response Time
7. No. of I/O Failures
8. I/O Efficiency

C.4 TASK DETAIL - EVENT SUMMARY

HEADER:

1. Task No.
2. Time
3. DISPLAY NO. 1 Detail

SELECTION:

1. None

HORIZONTAL

EVENT

1 - - - n

VERTICAL:

1. Event No.
2. Event Type [two character mnemonic]
3. Execution Time
4. Event Efficiency
5. Queue Size [for Type 2]
6. Initiating Mode [for Type 1: E = Elaborated; O = Task Object]
7. Initiating Task [for Type 1, Mode = O]
8. Current Event Indicator [mark current event with *]

C.5 QUEUE SUMMARY

HEADER:

1. Selection
2. Time

SELECTION:

1. All Queues
2. Selected Tasks
3. Selected Tasks and Events
4. Selected Events

HORIZONTAL:

QUEUE

1 - - - n

VERTICAL:

1. Task No.
2. Task Status
3. Task Efficiency-1
4. Task Efficiency-2
5. Event No.
6. Event Efficiency
7. Queue Type [Interrupt or Entry]
8. Average Wait Time
9. Queue Size

C.6 QUEUE HISTOGRAM

HEADER:

1. Selection
2. Time

SELECTION:

1. Same as Queue Summary Display

HORIZONTAL:

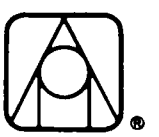
TIME CELL	INPUT NO.	OUTPUT NO. (INPUT NO.)	QUEUE 1	- - -	n

VERTICAL:

1. Task No. [Under QUEUE 1 - - - n]
2. Event No. [Under QUEUE 1 - - - n]
3. Time Range [Under TIME CELL]
4. Input No. [Enter Input Nos. (repeated vertically) under INPUT NO.]
5. Output No. [Enter Output Nos. and "(Input No.)" (repeated vertically) under OUTPUT NO.]
6. Queue Size [Enter number of items in queue per indicated time range under QUEUE 1 - - - n]

QUEUE HISTOGRAM (Example)

TIME CELL	INPUT NO.	OUTPUT NO. (INPUT NO.)	QUEUE 1	2	3	4
Task			2	3	3	5
Event			11	15	18	15
0 - 10			0	0	1	0
11 - 20	100 101		1	0	1	1
21 - 30	101 104		2	1	0	1
31 - 40		200 (100)	3	2	1	1



END
FILMED
4-86
DTIC